Annotated Ada Reference Manual

2022 Edition

Language and Standard Libraries

Copyright © 1992, 1993, 1994, 1995 Intermetrics, Inc.

Copyright © 2000 The MITRE Corporation, Inc.

Copyright © 2004, 2005, 2006 AXE Consultants

Copyright © 2004, 2005, 2006 Ada-Europe

Copyright © 2008, 2009, 2010, 2011, 2012, 2013, 2014, 2015, 2016, 2017, 2018, 2019, 2020, 2021, 2022, 2023 AXE Consultants

Ada Reference Manual - Language and Standard Libraries

Copyright © 1992, 1993, 1994, 1995, Intermetrics, Inc.

This copyright is assigned to the U.S. Government. All rights reserved.

This document may be copied, in whole or in part, in any form or by any means, as is or with alterations, provided that (1) alterations are clearly marked as alterations and (2) this copyright notice is included unmodified in any copy. Compiled copies of standard library units and examples need not contain this copyright notice so long as the notice is included in all copies of source code and documentation.

Technical Corrigendum 1

Copyright © 2000, The MITRE Corporation. All Rights Reserved.

This document may be copied, in whole or in part, in any form or by any means, as is, or with alterations, provided that (1) alterations are clearly marked as alterations and (2) this copyright notice is included unmodified in any copy. Any other use or distribution of this document is prohibited without the prior express permission of MITRE.

You use this document on the condition that you indemnify and hold harmless MITRE, its Board of Trustees, officers, agents, and employees, from any and all liability or damages to yourself or your hardware or software, or third parties, including attorneys' fees, court costs, and other related costs and expenses, arising out of your use of this document irrespective of the cause of said liability.

MITRE MAKES THIS DOCUMENT AVAILABLE ON AN "AS IS" BASIS AND MAKES NO WARRANTY, EXPRESS OR IMPLIED, AS TO THE ACCURACY, CAPABILITY, EFFICIENCY MERCHANTABILITY, OR FUNCTIONING OF THIS DOCUMENT. IN NO EVENT WILL MITRE BE LIABLE FOR ANY GENERAL, CONSEQUENTIAL, INDIRECT, INCIDENTAL, EXEMPLARY, OR SPECIAL DAMAGES, EVEN IF MITRE HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Amendment 1

Copyright © 2004, 2005, 2006, 2007, AXE Consultants. All Rights Reserved.

This document may be copied, in whole or in part, in any form or by any means, as is, or with alterations, provided that (1) alterations are clearly marked as alterations and (2) this copyright notice is included unmodified in any copy. Any other use or distribution of this document is prohibited without the prior express permission of AXE.

You use this document on the condition that you indemnify and hold harmless AXE, its board, officers, agents, and employees, from any and all liability or damages to yourself or your hardware or software, or third parties, including attorneys' fees, court costs, and other related costs and expenses, arising out of your use of this document irrespective of the cause of said liability.

AXE MAKES THIS DOCUMENT AVAILABLE ON AN "AS IS" BASIS AND MAKES NO WARRANTY, EXPRESS OR IMPLIED, AS TO THE ACCURACY, CAPABILITY, EFFICIENCY MERCHANTABILITY, OR FUNCTIONING OF THIS DOCUMENT. IN NO EVENT WILL AXE BE LIABLE FOR ANY GENERAL, CONSEQUENTIAL, INDIRECT, INCIDENTAL, EXEMPLARY, OR SPECIAL DAMAGES, EVEN IF AXE HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Third Edition (Ada 2012)

Copyright © 2008, 2009, 2010, 2011, 2012 AXE Consultants. All Rights Reserved.

This document may be copied, in whole or in part, in any form or by any means, as is, or with alterations, provided that (1) alterations are clearly marked as alterations and (2) this copyright notice is included unmodified in any copy. Any other use or distribution of this document is prohibited without the prior express permission of AXE.

You use this document on the condition that you indemnify and hold harmless AXE, its board, officers, agents, and employees, from any and all liability or damages to yourself or your hardware or software, or third parties, including attorneys' fees, court costs, and other related costs and expenses, arising out of your use of this document irrespective of the cause of said liability.

AXE MAKES THIS DOCUMENT AVAILABLE ON AN "AS IS" BASIS AND MAKES NO WARRANTY, EXPRESS OR IMPLIED, AS TO THE ACCURACY, CAPABILITY, EFFICIENCY MERCHANTABILITY, OR FUNCTIONING OF THIS DOCUMENT. IN NO EVENT WILL AXE BE LIABLE FOR ANY GENERAL, CONSEQUENTIAL, INDIRECT, INCIDENTAL, EXEMPLARY, OR SPECIAL DAMAGES, EVEN IF AXE HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Technical Corrigendum 1 for Ada 2012

Copyright © 2013, 2014, 2015, 2016 AXE Consultants. All Rights Reserved.

This document may be copied, in whole or in part, in any form or by any means, as is, or with alterations, provided that (1) alterations are clearly marked as alterations and (2) this copyright notice is included unmodified in any copy. Any other use or distribution of this document is prohibited without the prior express permission of AXE.

You use this document on the condition that you indemnify and hold harmless AXE, its board, officers, agents, and employees, from any and all liability or damages to yourself or your hardware or software, or third parties, including attorneys' fees, court costs, and other related costs and expenses, arising out of your use of this document irrespective of the cause of said liability.

AXE MAKES THIS DOCUMENT AVAILABLE ON AN "AS IS" BASIS AND MAKES NO WARRANTY, EXPRESS OR IMPLIED, AS TO THE ACCURACY, CAPABILITY, EFFICIENCY MERCHANTABILITY, OR FUNCTIONING OF THIS DOCUMENT. IN NO EVENT WILL AXE BE LIABLE FOR ANY GENERAL, CONSEQUENTIAL, INDIRECT, INCIDENTAL, EXEMPLARY, OR SPECIAL DAMAGES, EVEN IF AXE HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Fourth Edition (Ada 2022)

Copyright © 2016, 2017, 2018, 2019, 2020, 2021, 2022, 2023 AXE Consultants. All Rights Reserved.

This document may be copied, in whole or in part, in any form or by any means, as is, or with alterations, provided that (1) alterations are clearly marked as alterations and (2) this copyright notice is included

unmodified in any copy. Any other use or distribution of this document is prohibited without the prior express permission of AXE.

You use this document on the condition that you indemnify and hold harmless AXE, its board, officers, agents, and employees, from any and all liability or damages to yourself or your hardware or software, or third parties, including attorneys' fees, court costs, and other related costs and expenses, arising out of your use of this document irrespective of the cause of said liability.

AXE MAKES THIS DOCUMENT AVAILABLE ON AN "AS IS" BASIS AND MAKES NO WARRANTY, EXPRESS OR IMPLIED, AS TO THE ACCURACY, CAPABILITY, EFFICIENCY MERCHANTABILITY, OR FUNCTIONING OF THIS DOCUMENT. IN NO EVENT WILL AXE BE LIABLE FOR ANY GENERAL, CONSEQUENTIAL, INDIRECT, INCIDENTAL, EXEMPLARY, OR SPECIAL DAMAGES, EVEN IF AXE HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Ada 2005 Consolidated Reference Manual

Copyright © 2004, 2005, 2006, Ada-Europe.

This document may be copied, in whole or in part, in any form or by any means, as is, or with alterations, provided that (1) alterations are clearly marked as alterations and (2) this copyright notice is included unmodified in any copy. Any other use or distribution of this document is prohibited without the prior express permission of Ada-Europe.

You use this document on the condition that you indemnify and hold harmless Ada-Europe and its Board from any and all liability or damages to yourself or your hardware or software, or third parties, including attorneys' fees, court costs, and other related costs and expenses, arising out of your use of this document irrespective of the cause of said liability.

ADA-EUROPE MAKES THIS DOCUMENT AVAILABLE ON AN "AS IS" BASIS AND MAKES NO WARRANTY, EXPRESS OR IMPLIED, AS TO THE ACCURACY, CAPABILITY, EFFICIENCY MERCHANTABILITY, OR FUNCTIONING OF THIS DOCUMENT. IN NO EVENT WILL ADA-EUROPE BE LIABLE FOR ANY GENERAL, CONSEQUENTIAL, INDIRECT, INCIDENTAL, EXEMPLARY, OR SPECIAL DAMAGES, EVEN IF ADA-EUROPE HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Table of Contents

Table of Contents i
Foreword xiii
Introduction xv
1 General 1
1.1 Scope 2
1.1.1 Extent 2
1.1.2 Structure 3
1.1.3 Conformity of an Implementation 10
1.1.4 Method of Description and Syntax Notation 13
1.1.5 Classification of Errors 16
1.2 Normative References 18
1.2.1 Bibliography 18
1.3 Terms and Definitions 19
1.3.1 Types, Objects, and their Properties 20
1.3.2 Subprograms and their Properties 23
1.3.3 Other Syntactic Constructs 24
1.3.4 Runtime Actions 26
1.3.5 Exceptional Situations 26
2 Lexical Elements 29
2.1 Character Set 29
2.2 Lexical Elements, Separators, and Delimiters 33
2.3 Identifiers 35
2.4 Numeric Literals 37
2.4.1 Decimal Literals 37
2.4.2 Based Literals 38
2.5 Character Literals 39
2.6 String Literals 39
2.7 Comments 40
2.8 Pragmas 40
2.9 Reserved Words 45
3 Declarations and Types 49
3.1 Declarations 49
3.2 Types and Subtypes 52
3.2.1 Type Declarations 56
3.2.2 Subtype Declarations 58
3.2.3 Classification of Operations 60
3.2.4 Subtype Predicates 61
3.3 Objects and Named Numbers 66
3.3.1 Object Declarations 70
3.3.2 Number Declarations 75
3.4 Derived Types and Classes 76
3.4.1 Derivation Classes 83
3.5 Scalar Types 85
3.5.1 Enumeration Types 92

3.5.2 Character Types 93
3.5.3 Boolean Types 95
3.5.4 Integer Types 96
3.5.5 Operations of Discrete Types 100
3.5.6 Real Types 102
3.5.7 Floating Point Types 103
3.5.8 Operations of Floating Point Types 106
3.5.9 Fixed Point Types 106
3.5.10 Operations of Fixed Point Types 109
3.6 Array Types 110
3.6.1 Index Constraints and Discrete Ranges 114
3.6.2 Operations of Array Types 115
3.6.3 String Types 116
3.7 Discriminants 117
3.7.1 Discriminant Constraints 123
3.7.2 Operations of Discriminated Types 126
3.8 Record Types 127
3.8.1 Variant Parts and Discrete Choices 130
3.9 Tagged Types and Type Extensions 134
3.9.1 Type Extensions 142
3.9.2 Dispatching Operations of Tagged Types 145
3.9.3 Abstract Types and Subprograms 150
3.9.4 Interface Types 156
3.10 Access Types 160
3.10.1 Incomplete Type Declarations 165
3.10.2 Operations of Access Types 171
3.11 Declarative Parts 186
3.11.1 Completions of Declarations 188
4 Names and Expressions 191
4.1 Names 191
4.1.1 Indexed Components 193
4.1.2 Slices 194
4.1.3 Selected Components 195
4.1.4 Attributes 198
4.1.5 User-Defined References 201
4.1.6 User-Defined Indexing 203
4.2 Literals 205
4.2.1 User-Defined Literals 207
4.3 Aggregates 209
4.3.1 Record Aggregates 211
4.3.2 Extension Aggregates 216
4.3.3 Array Aggregates 218
4.3.4 Delta Aggregates 225
4.3.5 Container Aggregates 228
4.4 Expressions 234
4.5 Operators and Expression Evaluation 238
4.5.1 Logical Operators and Short-circuit Control Forms 239
4.5.2 Relational Operators and Membership Tests 241
4.5.3 Binary Adding Operators 248

4.5.4 Unary Adding Operators 250
4.5.5 Multiplying Operators 250
4.5.6 Highest Precedence Operators 253
4.5.7 Conditional Expressions 254
4.5.8 Quantified Expressions 257
4.5.9 Declare Expressions 258
4.5.10 Reduction Expressions 260
4.6 Type Conversions 264
4.7 Qualified Expressions 274
4.8 Allocators 275
4.9 Static Expressions and Static Subtypes 280
4.9.1 Statically Matching Constraints and Subtypes 288
4.10 Image Attributes 290
5 Statements 297
5.1 Simple and Compound Statements - Sequences of Statements 297
5.2 Assignment Statements 300
5.2.1 Target Name Symbols 303
5.3 If Statements 304
5.4 Case Statements 304
5.5 Loop Statements 307
5.5.1 User-Defined Iterator Types 312
5.5.2 Generalized Loop Iteration 315
5.5.3 Procedural Iterators 319
5.6 Block Statements 323
5.6.1 Parallel Block Statements 324
5.7 Exit Statements 326
5.8 Goto Statements 327
6 Subprograms 329
6.1 Subprogram Declarations 329
6.1.1 Preconditions and Postconditions 333
6.1.2 The Global and Global'Class Aspects 344
6.2 Formal Parameter Modes 348
6.3 Subprogram Bodies 350
6.3.1 Conformance Rules 352
6.3.2 Inline Expansion of Subprograms 357
6.4 Subprogram Calls 358
6.4.1 Parameter Associations 360
6.5 Return Statements 368
6.5.1 Nonreturning Subprograms 376
6.6 Overloading of Operators 377
6.7 Null Procedures 378
6.8 Expression Functions 379
7 Packages 383
7.1 Package Specifications and Declarations 383
7.2 Package Bodies 385
7.3 Private Types and Private Extensions 386
7.3.1 Private Operations 393
7.3.2 Type Invariants 398

7.3.3 Default Initial Conditions 404
7.3.4 Stable Properties of a Type 405
7.4 Deferred Constants 409
7.5 Limited Types 411
7.6 Assignment and Finalization 415
7.6.1 Completion and Finalization 422
8 Visibility Rules 431
8.1 Declarative Region 431
8.2 Scope of Declarations 433
8.3 Visibility 436
8.3.1 Overriding Indicators 444
8.4 Use Clauses 445
8.5 Renaming Declarations 447
8.5.1 Object Renaming Declarations 448
8.5.2 Exception Renaming Declarations 452
8.5.3 Package Renaming Declarations 452
8.5.4 Subprogram Renaming Declarations 453
8.5.5 Generic Renaming Declarations 457
8.6 The Context of Overload Resolution 458
9 Tasks and Synchronization 467
9.1 Task Units and Task Objects 468
9.2 Task Execution - Task Activation 472
9.3 Task Dependence - Termination of Tasks 474
9.4 Protected Units and Protected Objects 475
9.5 Intertask Communication 481
9.5.1 Protected Subprograms and Protected Actions 489
9.5.2 Entries and Accept Statements 491
9.5.3 Entry Calls 496
9.5.4 Requeue Statements 500
9.6 Delay Statements, Duration, and Time 503
9.6.1 Formatting, Time Zones, and other operations for Time 507
9.7 Select Statements 515
9.7.1 Selective Accept 516
9.7.2 Timed Entry Calls 518
9.7.3 Conditional Entry Calls 519
9.7.4 Asynchronous Transfer of Control 520
9.8 Abort of a Task - Abort of a Sequence of Statements 521
9.9 Task and Entry Attributes 524
9.10 Shared Variables 524
9.10.1 Conflict Check Policies 527
9.11 Example of Tasking and Synchronization 529
10 Program Structure and Compilation Issues 533
10.1 Separate Compilation 533
10.1.1 Compilation Units - Library Units 534
10.1.2 Context Clauses - With Clauses 541
10.1.3 Subunits of Compilation Units 546
10.1.4 The Compilation Process 548
10.1.5 Pragmas and Program Units 551

10.1.6 Environment-Level Visibility Rules 552
10.2 Program Execution 553
10.2.1 Elaboration Control 558
11 Exceptions 567
11.1 Exception Declarations 567
11.2 Exception Handlers 568
11.3 Raise Statements and Raise Expressions 570
11.4 Exception Handling 572
11.4.1 The Package Exceptions 573
11.4.2 Pragmas Assert and Assertion_Policy 578
11.4.3 Example of Exception Handling 582
11.5 Suppressing Checks 584
11.6 Exceptions and Optimization 589
12 Generic Units 593
12.1 Generic Declarations 593
12.2 Generic Bodies 595
12.3 Generic Instantiation 596
12.4 Formal Objects 605
12.5 Formal Types 607
12.5.1 Formal Private and Derived Types 611
12.5.2 Formal Scalar Types 616
12.5.3 Formal Array Types 617
12.5.4 Formal Access Types 618
12.5.5 Formal Interface Types 619
12.6 Formal Subprograms 620
12.7 Formal Packages 625
12.8 Example of a Generic Package 628
13 Representation Issues 631
13.1 Operational and Representation Aspects 631
13.1.1 Aspect Specifications 644
13.2 Packed Types 651
13.3 Operational and Representation Attributes 653
13.4 Enumeration Representation Clauses 669
13.5 Record Layout 672
13.5.1 Record Representation Clauses 672
13.5.2 Storage Place Attributes 676
13.5.3 Bit Ordering 677
13.6 Change of Representation 678
13.7 The Package System 679
13.7.1 The Package System.Storage_Elements 682
13.7.2 The Package System.Address_To_Access_Conversions 684
13.8 Machine Code Insertions 684
13.9 Unchecked Type Conversions 686
13.9.1 Data Validity 688
13.9.2 The Valid Attribute 691
13.10 Unchecked Access Value Creation 692
13.11 Storage Management 692
13.11.1 Storage Allocation Attributes 700

13.11.2 Unchecked Storage Deallocation 701
13.11.3 Default Storage Pools 704
13.11.4 Storage Subpools 706
13.11.5 Subpool Reclamation 710
13.11.6 Storage Subpool Example 711
13.12 Pragma Restrictions and Pragma Profile 713
13.12.1 Language-Defined Restrictions and Profiles 715
13.13 Streams 718
13.13.1 The Streams Subsystem 719
13.13.2 Stream-Oriented Attributes 722
13.14 Freezing Rules 732

The Standard Libraries 743

Annex A (normative) **Predefined Language Environment** 745

A.1 The Package Standard 750
A.2 The Package Ada 755
A.3 Character Handling 755
A.3.1 The Packages Characters, Wide_Characters, and Wide_Wide_Characters 756
A.3.2 The Package Characters.Handling 756
A.3.3 The Package Characters.Latin_1 760
A.3.4 The Package Characters.Conversions 765
A.3.5 The Package Wide_Characters.Handling 767
A.3.6 The Package Wide_Wide_Characters.Handling 770
A.4 String Handling 770
A.4.1 The Package Strings 771
A.4.2 The Package Strings.Maps 771
A.4.3 Fixed-Length String Handling 775
A.4.4 Bounded-Length String Handling 784
A.4.5 Unbounded-Length String Handling 792
A.4.6 String-Handling Sets and Mappings 798
A.4.7 Wide_String Handling 799
A.4.8 Wide_Wide_String Handling 802
A.4.9 String Hashing 805
A.4.10 String Comparison 806
A.4.11 String Encoding 808
A.4.12 Universal Text Buffers 813
A.5 The Numerics Packages 816
A.5.1 Elementary Functions 816
A.5.2 Random Number Generation 820
A.5.3 Attributes of Floating Point Types 827
A.5.4 Attributes of Fixed Point Types 833
A.5.5 Big Numbers 834
A.5.6 Big Integers 834
A.5.7 Big Reals 836
A.6 Input-Output 838
A.7 External Files and File Objects 839
A.8 Sequential and Direct Files 840
A.8.1 The Generic Package Sequential_IO 841
A.8.2 File Management 843
A.8.3 Sequential Input-Output Operations 846

A.8.4 The Generic Package Direct_IO 846
A.8.5 Direct Input-Output Operations 848
A.9 The Generic Package Storage_IO 849
A.10 Text Input-Output 850
A.10.1 The Package Text_IO 852
A.10.2 Text File Management 859
A.10.3 Default Input, Output, and Error Files 859
A.10.4 Specification of Line and Page Lengths 861
A.10.5 Operations on Columns, Lines, and Pages 862
A.10.6 Get and Put Procedures 865
A.10.7 Input-Output of Characters and Strings 866
A.10.8 Input-Output for Integer Types 869
A.10.9 Input-Output for Real Types 870
A.10.10 Input-Output for Enumeration Types 873
A.10.11 Input-Output for Bounded Strings 875
A.10.12 Input-Output for Unbounded Strings 876
A.11 Wide Text Input-Output and Wide Wide Text Input-Output 877
A.12 Stream Input-Output 878
A.12.1 The Package Streams.Stream_IO 878
A.12.2 The Package Text_IO.Text_Streams 883
A.12.3 The Package Wide_Text_IO.Text_Streams 883
A.12.4 The Package Wide_Wide_Text_IO.Text_Streams 883
A.13 Exceptions in Input-Output 884
A.14 File Sharing 885
A.15 The Package Command_Line 886
A.15.1 The Packages Wide_Command_Line and Wide_Wide_Command_Line 887
A.16 The Package Directories 887
A.16.1 The Package Directories.Hierarchical_File_Names 899
A.16.2 The Packages Wide_Directories and Wide_Wide_Directories 902
A.17 The Package Environment_Variables 902
A.17.1 The Packages Wide_Environment_Variables and Wide_Wide_Environment_Variables 904
A.18 Containers 905
A.18.1 The Package Containers 909
A.18.2 The Generic Package Containers.Vectors 910
A.18.3 The Generic Package Containers.Doubly_Linked_Lists 950
A.18.4 Maps 977
A.18.5 The Generic Package Containers.Hashed_Maps 989
A.18.6 The Generic Package Containers.Ordered_Maps 1001
A.18.7 Sets 1014
A.18.8 The Generic Package Containers.Hashed_Sets 1028
A.18.9 The Generic Package Containers.Ordered_Sets 1039
A.18.10 The Generic Package Containers.Multiway_Trees 1052
A.18.11 The Generic Package Containers.Indefinite_Vectors 1093
A.18.12 The Generic Package Containers.Indefinite_Doubly_Linked_Lists 1094
A.18.13 The Generic Package Containers.Indefinite_Hashed_Maps 1095
A.18.14 The Generic Package Containers.Indefinite_Ordered_Maps 1096
A.18.15 The Generic Package Containers.Indefinite_Hashed_Sets 1097
A.18.16 The Generic Package Containers.Indefinite_Ordered_Sets 1098
A.18.17 The Generic Package Containers.Indefinite_Multiway_Trees 1099

A.18.18 The Generic Package Containers.Indefinite_Holders 1100
A.18.19 The Generic Package Containers.Bounded_Vectors 1106
A.18.20 The Generic Package Containers.Bounded_Doubly_Linked_Lists 1108
A.18.21 The Generic Package Containers.Bounded_Hashed_Maps 1111
A.18.22 The Generic Package Containers.Bounded_Ordered_Maps 1113
A.18.23 The Generic Package Containers.Bounded_Hashed_Sets 1116
A.18.24 The Generic Package Containers.Bounded_Ordered_Sets 1118
A.18.25 The Generic Package Containers.Bounded_Multiway_Trees 1120
A.18.26 Array Sorting 1123
A.18.27 The Generic Package Containers.Synchronized_Queue_Interfaces 1125
A.18.28 The Generic Package Containers.Unbounded_Synchronized_Queues 1127
A.18.29 The Generic Package Containers.Bounded_Synchronized_Queues 1128
A.18.30 The Generic Package Containers.Unbounded_Priority_Queues 1129
A.18.31 The Generic Package Containers.Bounded_Priority_Queues 1130
A.18.32 The Generic Package Containers.Bounded_Indefinite_Holders 1131
A.18.33 Example of Container Use 1133
A.19 The Package Locales 1135
Annex B (normative) Interface to Other Languages 1137
B.1 Interfacing Aspects 1137
B.2 The Package Interfaces 1143
B.3 Interfacing with C and C++ 1145
B.3.1 The Package Interfaces.C.Strings 1154
B.3.2 The Generic Package Interfaces.C.Pointers 1158
B.3.3 Unchecked Union Types 1161
B.4 Interfacing with COBOL 1163
B.5 Interfacing with Fortran 1170
Annex C (normative) Systems Programming 1175
C.1 Access to Machine Operations 1175
C.2 Required Representation Support 1176
C.3 Interrupt Support 1176
C.3.1 Protected Procedure Handlers 1179
C.3.2 The Package Interrupts 1182
C.4 Preelaboration Requirements 1185
C.5 Aspect Discard_Names 1186
C.6 Shared Variable Control 1188
C.6.1 The Package System.Atomic_Operations 1194
C.6.2 The Package System.Atomic_Operations.Exchange 1194
C.6.3 The Package System.Atomic_Operations.Test_and_Set 1195
C.6.4 The Package System.Atomic_Operations.Integer_Arithmetic 1196
C.6.5 The Package System.Atomic_Operations.Modular_Arithmetic 1197
C.7 Task Information 1198
C.7.1 The Package Task_Identification 1199
C.7.2 The Package Task_Attributes 1201
C.7.3 The Package Task_Termination 1204
Annex D (normative) Real-Time Systems 1207
D.1 Task Priorities 1207
D.2 Priority Scheduling 1210
D.2.1 The Task Dispatching Model 1210

D.2.2 Task Dispatching Pragmas 1213
D.2.3 Preemptive Dispatching 1215
D.2.4 Non-Preemptive Dispatching 1217
D.2.5 Round Robin Dispatching 1218
D.2.6 Earliest Deadline First Dispatching 1220
D.3 Priority Ceiling Locking 1224
D.4 Entry Queuing Policies 1227
D.4.1 Admission Policies 1230
D.5 Dynamic Priorities 1231
D.5.1 Dynamic Priorities for Tasks 1231
D.5.2 Dynamic Priorities for Protected Objects 1233
D.6 Preemptive Abort 1235
D.7 Tasking Restrictions 1236
D.8 Monotonic Time 1241
D.9 Delay Accuracy 1245
D.10 Synchronous Task Control 1246
D.10.1 Synchronous Barriers 1248
D.11 Asynchronous Task Control 1249
D.12 Other Optimizations and Determinism Rules 1250
D.13 The Ravenscar and Jorvik Profiles 1251
D.14 Execution Time 1254
D.14.1 Execution Time Timers 1257
D.14.2 Group Execution Time Budgets 1259
D.14.3 Execution Time of Interrupt Handlers 1262
D.15 Timing Events 1262
D.16 Multiprocessor Implementation 1265
D.16.1 Multiprocessor Dispatching Domains 1267
Annex E (normative) **Distributed Systems** 1271
E.1 Partitions 1271
E.2 Categorization of Library Units 1273
E.2.1 Shared Passive Library Units 1274
E.2.2 Remote Types Library Units 1276
E.2.3 Remote Call Interface Library Units 1279
E.3 Consistency of a Distributed System 1282
E.4 Remote Subprogram Calls 1283
E.4.1 Asynchronous Remote Calls 1287
E.4.2 Example of Use of a Remote Access-to-Class-Wide Type 1288
E.5 Partition Communication Subsystem 1290
Annex F (normative) **Information Systems** 1295
F.1 Machine_Radix Attribute Definition Clause 1295
F.2 The Package Decimal 1296
F.3 Edited Output for Decimal Types 1297
F.3.1 Picture String Formation 1298
F.3.2 Edited Output Generation 1302
F.3.3 The Package Text_IO.Editing 1306
F.3.4 The Package Wide_Text_IO.Editing 1310
F.3.5 The Package Wide_Wide_Text_IO.Editing 1310
Annex G (normative) **Numerics** 1311

G.1 Complex Arithmetic ... **1311**
G.1.1 Complex Types ... **1312**
G.1.2 Complex Elementary Functions ... **1317**
G.1.3 Complex Input-Output ... **1322**
G.1.4 The Package Wide_Text_IO.Complex_IO ... **1325**
G.1.5 The Package Wide_Wide_Text_IO.Complex_IO ... **1325**
G.2 Numeric Performance Requirements ... **1325**
G.2.1 Model of Floating Point Arithmetic ... **1326**
G.2.2 Model-Oriented Attributes of Floating Point Types ... **1328**
G.2.3 Model of Fixed Point Arithmetic ... **1330**
G.2.4 Accuracy Requirements for the Elementary Functions ... **1333**
G.2.5 Performance Requirements for Random Number Generation ... **1335**
G.2.6 Accuracy Requirements for Complex Arithmetic ... **1336**
G.3 Vector and Matrix Manipulation ... **1339**
G.3.1 Real Vectors and Matrices ... **1339**
G.3.2 Complex Vectors and Matrices ... **1345**
Annex H (normative) **High Integrity Systems** ... **1359**
H.1 Pragma Normalize_Scalars ... **1359**
H.2 Documentation of Implementation Decisions ... **1360**
H.3 Reviewable Object Code ... **1361**
H.3.1 Pragma Reviewable ... **1361**
H.3.2 Pragma Inspection_Point ... **1363**
H.4 High Integrity Restrictions ... **1364**
H.4.1 Aspect No_Controlled_Parts ... **1370**
H.5 Pragma Detect_Blocking ... **1370**
H.6 Pragma Partition_Elaboration_Policy ... **1371**
H.7 Extensions to Global and Global'Class Aspects ... **1372**
H.7.1 The Use_Formal and Dispatching Aspects ... **1373**
Annex J (normative) **Obsolescent Features** ... **1377**
J.1 Renamings of Library Units ... **1377**
J.2 Allowed Replacements of Characters ... **1378**
J.3 Reduced Accuracy Subtypes ... **1378**
J.4 The Constrained Attribute ... **1380**
J.5 ASCII ... **1380**
J.6 Numeric_Error ... **1381**
J.7 At Clauses ... **1381**
J.7.1 Interrupt Entries ... **1381**
J.8 Mod Clauses ... **1383**
J.9 The Storage_Size Attribute ... **1384**
J.10 Specific Suppression of Checks ... **1384**
J.11 The Class Attribute of Untagged Incomplete Types ... **1385**
J.12 Pragma Interface ... **1385**
J.13 Dependence Restriction Identifiers ... **1386**
J.14 Character and Wide_Character Conversion Functions ... **1386**
J.15 Aspect-related Pragmas ... **1386**
J.15.1 Pragma Inline ... **1388**
J.15.2 Pragma No_Return ... **1389**
J.15.3 Pragma Pack ... **1389**
J.15.4 Pragma Storage_Size ... **1390**

J.15.5 Interfacing Pragmas **1390**
J.15.6 Pragma Unchecked_Union **1391**
J.15.7 Pragmas Interrupt_Handler and Attach_Handler **1392**
J.15.8 Shared Variable Pragmas **1393**
J.15.9 Pragma CPU **1393**
J.15.10 Pragma Dispatching_Domain **1394**
J.15.11 Pragmas Priority and Interrupt_Priority **1395**
J.15.12 Pragma Relative_Deadline **1395**
J.15.13 Pragma Asynchronous **1396**
J.15.14 Elaboration Control Pragmas **1396**
J.15.15 Distribution Pragmas **1397**

Annex K (informative) **Language-Defined Aspects and Attributes** **1399**

K.1 Language-Defined Aspects **1399**
K.2 Language-Defined Attributes **1403**

Annex L (informative) **Language-Defined Pragmas** **1421**

Annex M (informative) **Summary of Documentation Requirements** **1425**

M.1 Specific Documentation Requirements **1425**
M.2 Implementation-Defined Characteristics **1427**
M.3 Implementation Advice **1434**

Annex N (informative) **Glossary** **1445**

Annex P (informative) **Syntax Summary** **1447**

P.1 Syntax Rules **1447**
P.2 Syntax Cross Reference **1469**

Annex Q (informative) **Language-Defined Entities** **1483**

Q.1 Language-Defined Packages **1483**
Q.2 Language-Defined Types and Subtypes **1486**
Q.3 Language-Defined Subprograms **1490**
Q.4 Language-Defined Exceptions **1503**
Q.5 Language-Defined Objects **1504**

Index **1509**

Foreword

This document is the Annotated Ada Reference Manual. The International Standard for the programming language Ada is ISO/IEC 8652:2023(E). The International Standard is derived from the Ada Reference Manual, with various non-normative changes. In particular, the International Standard numbers clauses differently, omits the annotations and paragraph numbers, eliminates the Acknowledgements, and modifies various front matter such as the Title page and the Foreword. 0.1/5

Discussion: New ISO requirements necessitated changing the numbers of all of the numbered clauses and subclauses in the International Standard. The Ada Reference Manual retains its traditional numbering of clauses and subclauses as it is common for existing material (both printed and electronic) to refer to subclauses just by number. 0.a/5

The Ada Working Group ISO/IEC JTC 1/SC 22/WG 9 is tasked by ISO with the work item to interpret and maintain the International Standard and to produce Technical Corrigenda, as appropriate. The technical work on the International Standard is performed by the Ada Rapporteur Group (ARG) of WG 9. 0.2/4

AXE Consultants produces the Ada Reference Manual in consultation with the ARG, along with drafts of other documents as needed. ISO/IEC documents often list the individual changes that need to be made to the text of a Standard, rather than simply updating the document. As such, an International Standard is often found in several parts, while the Ada Reference Manual is always a single document. 0.3/2

In June 2016, WG 9 approved a tentative schedule for the preparation of an Amendment or Revision to the International Standard, with a delivery no earlier than 2018. In July 2019, WG 9 approved an additional review and prototyping period for this revision, extending the delivery to no earlier than late 2020. The draft standard was delivered to WG 9 in July 2021, and the Standard was published as ISO/IEC 8652:2023 in May 2023. 0.5/5

{*AI12-0313-1*} {*AI12-0441-1*} This Ada Reference Manual replaces the edition of 2012. It modifies the previous edition by making changes and additions that improve the capability of the language and the reliability of programs written in the language. 1/5

{*AI12-0441-1*} Significant changes in this edition are as follows: 2/5

- {*AI12-0313-1*} {*AI12-0441-1*} Improved support for parallel execution is provided via the introduction of parallel loops, parallel blocks, parallel container iteration, and parallel reduction. 3/5
- {*AI12-0313-1*} {*AI12-0441-1*} More precise specification of subprogram interfaces is supported via the new aspects Global, Global'Class, and Nonblocking. The Global aspects, in particular, help to determine whether two constructs can safely execute in parallel. 4/5
- {*AI12-0313-1*} {*AI12-0441-1*} Pre and Post aspects can now be specified for access-to-subprogram types and for generic formal subprograms; a postcondition for the default initialization of a type can be specified using the new Default_Initial_Condition aspect. 5/5
- {*AI12-0313-1*} {*AI12-0441-1*} The behavior of many predefined container operations is now more precisely specified by using pre- and postcondition specifications instead of English descriptions; a restricted (“stable”) view for most containers is introduced to support more efficient iteration. 6/5
- {*AI12-0313-1*} {*AI12-0441-1*} More flexible uses of static expressions are supported via the introduction of static expression functions along with fewer restrictions on static strings. 7/5
- {*AI12-0313-1*} {*AI12-0441-1*} The Image attribute is supported for nonscalar types, and a user-specifiable attribute Put_Image is provided, which determines the value of the Image attribute for a user-defined type. 8/5

9/5 • {*AI12-0313-1*} {*AI12-0441-1*} The use of numeric and string literals is generalized to allow their use with other categories of types, via the new aspects Integer_Literal, Real_Literal, and String_Literal.

10/5 • {*AI12-0313-1*} {*AI12-0441-1*} Array and record aggregates are made more flexible: index parameters are allowed in an array aggregate to define the components as a function of their array index; discriminants can be defined more flexibly within an aggregate for a variant record type.

11/5 • {*AI12-0313-1*} {*AI12-0441-1*} New types of aggregates are provided: delta aggregates to allow the construction of a new object by incremental updates to an existing object; container aggregates to allow construction of an object of a container type by directly specifying its elements.

12/5 • {*AI12-0313-1*} {*AI12-0441-1*} A shorthand is provided, using the token '@', to refer to the target of an assignment statement in the expression defining its new value.

13/5 • {*AI12-0313-1*} {*AI12-0441-1*} Declare expressions are provided that permit the definition and use of local constants or renamings, to allow a large expression to be simplified by defining common parts as named entities.

14/5 • {*AI12-0313-1*} {*AI12-0441-1*} Support for lightweight iteration is added via the introduction of procedural iterators.

15/5 • {*AI12-0313-1*} {*AI12-0441-1*} Support for the map-reduce programming strategy is added via the introduction of reduction expressions.

16/5 • {*AI12-0313-1*} {*AI12-0441-1*} For constructs that use iterators of any sort, a filter can be specified that restricts the elements produced by the iteration to those that satisfy the condition of the filter.

17/5 • {*AI12-0313-1*} {*AI12-0441-1*} Predefined packages supporting arbitrary-precision integer and real arithmetic are provided.

18/5 • {*AI12-0313-1*} {*AI12-0441-1*} The Jorvik profile is introduced to support hard real-time applications that want to go beyond the restrictions of the Ravenscar profile.

Introduction

This is the Annotated Ada Reference Manual. 1

Discussion: This document is the Annotated Ada Reference Manual (AARM). It contains the entire text of the Ada 2022 standard (ISO/IEC 8652:2023), plus various annotations. It is intended primarily for compiler writers, validation test writers, and other language lawyers. The annotations include detailed rationale for individual rules and explanations of some of the more arcane interactions among the rules. 1.a/5

Other available Ada documents include: 2

- {*AI95-00387-01*} {*AI05-0245-1*} Ada 2022 Overview. This gives an introduction to the changes and new features in Ada 2022. Programmers should read this overview before reading Reference Manual in depth. 3/5
- *This paragraph was deleted.* 4/1
- The Ada Reference Manual (RM). This directly corresponds to the International Standard — ISO/IEC 8652:2023. 5/3

Design Goals

{*AI95-00387-01*} {*AI12-0313-1*} Ada was originally designed with three overriding concerns: program reliability and maintenance, programming as a human activity, and efficiency. The 1995 revision to the language was designed to provide greater flexibility and extensibility, additional control over storage management and synchronization, and standardized packages oriented toward supporting important application areas, while at the same time retaining the original emphasis on reliability, maintainability, and efficiency. Subsequent editions, including this fourth edition, have provided further flexibility and added more standardized packages within the framework provided by the 1995 revision. 6/5

The need for languages that promote reliability and simplify maintenance is well established. Hence emphasis was placed on program readability over ease of writing. For example, the rules of the language require that program variables be explicitly declared and that their type be specified. Since the type of a variable is invariant, compilers can ensure that operations on variables are compatible with the properties intended for objects of the type. Furthermore, error-prone notations have been avoided, and the syntax of the language avoids the use of encoded forms in favor of more English-like constructs. Finally, the language offers support for separate compilation of program units in a way that facilitates program development and maintenance, and which provides the same degree of checking between units as within a unit. 7

Concern for the human programmer was also stressed during the design. Above all, an attempt was made to keep to a relatively small number of underlying concepts integrated in a consistent and systematic way while continuing to avoid the pitfalls of excessive involution. The design especially aims to provide language constructs that correspond intuitively to the normal expectations of users. 8

Like many other human activities, the development of programs is becoming ever more decentralized and distributed. Consequently, the ability to assemble a program from independently produced software components continues to be a central idea in the design. The concepts of packages, of private types, and of generic units are directly related to this idea, which has ramifications in many other aspects of the language. An allied concern is the maintenance of programs to match changing requirements; type extension and the hierarchical library enable a program to be modified while minimizing disturbance to existing tested and trusted components. 9

{*AI12-0313-1*} No language can avoid the problem of efficiency. Languages that require over-elaborate compilers, or that lead to the inefficient use of storage or execution time, force these inefficiencies on all machines and on all programs. Every construct of the language was examined in the light of present 10/5

implementation techniques. Any proposed construct whose implementation was unclear or that required excessive machine resources was rejected. Parallel constructs were introduced to simplify making safe and efficient use of modern multicore architectures.

Language Summary

11/5 {*AI12-0438-1*} {*AI12-0441-1*} An Ada program is composed of one or more program units. Program units can be subprograms (which define executable algorithms), packages (which define collections of entities), task units (which define concurrent computations), protected units (which define operations for the coordinated sharing of data between tasks), or generic units (which define parameterized forms of packages and subprograms). Each program unit normally consists of two parts: a specification, containing the information that is visible to other units, and a body, containing the implementation details, which are not visible to other units. Most program units can be compiled separately.

12 This distinction of the specification and body, and the ability to compile units separately, allows a program to be designed, written, and tested as a set of largely independent software components.

13/5 {*AI12-0446-1*} An Ada program will normally make use of a library of program units of general utility. The language provides means whereby individual organizations can construct their own libraries. All libraries are structured in a hierarchical manner; this enables the logical decomposition of a subsystem into individual components. The text of a separately compiled program unit names the library units it requires.

14 *Program Units*

15/5 {*AI12-0441-1*} A subprogram is the basic unit for expressing an algorithm. There are two kinds of subprograms: procedures and functions. A procedure is the means of invoking a series of actions. For example, it can read data, update variables, or produce some output. It can have parameters, to provide a controlled means of passing information between the procedure and the point of call. A function is the means of invoking the computation of a value. It is similar to a procedure, but in addition will return a result.

16 A package is the basic unit for defining a collection of logically related entities. For example, a package can be used to define a set of type declarations and associated operations. Portions of a package can be hidden from the user, thus allowing access only to the logical properties expressed by the package specification.

17/5 {*AI12-0441-1*} Subprogram and package units can be compiled separately and arranged in hierarchies of parent and child units giving fine control over visibility of the logical properties and their detailed implementation.

18/5 {*AI12-0441-1*} A task unit is the basic unit for defining a task whose sequence of actions can be executed concurrently with those of other tasks. Such tasks can be implemented on multicomputers, multiprocessors, or with interleaved execution on a single processor. A task unit can define either a single executing task or a task type permitting the creation of any number of similar tasks.

19/5 {*AI95-00114-01*} {*AI12-0441-1*} {*AI12-0446-1*} A protected unit is the basic unit for defining protected operations for the coordinated use of data shared between tasks. Simple mutual exclusion is provided automatically, and more elaborate sharing protocols can be defined. A protected operation can either be a subprogram or an entry. A protected entry specifies a Boolean expression (an entry barrier) that blocks the execution of the body until it evaluates to True. A protected unit can define a single protected object or a protected type permitting the creation of several similar objects.

Declarations and Statements 20

The body of a program unit generally contains two parts: a declarative part, which defines the logical entities to be used in the program unit, and a sequence of statements, which defines the execution of the program unit. 21

{*AI12-0441-1*} The declarative part associates names with declared entities. For example, a name can denote a type, a constant, a variable, or an exception. A declarative part also introduces the names and parameters of other nested subprograms, packages, task units, protected units, and generic units to be used in the program unit. 22/5

{*AI12-0454-1*} The sequence of statements describes a sequence of actions to be performed. The statements are executed in succession (unless a transfer of control causes execution to continue from another place). 23/5

An assignment statement changes the value of a variable. A procedure call invokes execution of a procedure after associating any actual parameters provided at the call with the corresponding formal parameters. 24

Case statements and if statements allow the selection of an enclosed sequence of statements based on the value of an expression or on the value of a condition. 25

{*AI12-0446-1*} The loop statement provides the basic iterative mechanism in the language. A loop statement specifies a sequence of statements that are executed repeatedly as directed by an iteration scheme, or until an exit statement is encountered. 26/5

A block statement comprises a sequence of statements preceded by the declaration of local entities used by the statements. 27

{*AI12-0119-1*} Certain statements are associated with concurrent execution. A delay statement delays the execution of a task for a specified duration or until a specified time. An entry call statement is written as a procedure call statement; it requests an operation on a task or on a protected object, blocking the caller until the operation can be performed. A called task can accept an entry call by executing a corresponding accept statement, which specifies the actions then to be performed as part of the rendezvous with the calling task. An entry call on a protected object is processed when the corresponding entry barrier evaluates to true, whereupon the body of the entry is executed. The requeue statement permits the provision of a service as a number of related activities with preference control. One form of the select statement allows a selective wait for one of several alternative rendezvous. Other forms of the select statement allow conditional or timed entry calls and the asynchronous transfer of control in response to some triggering event. Various parallel constructs, including parallel loops and parallel blocks, support the initiation of multiple logical threads of control designed to execute in parallel when multiple processors are available. 28/5

{*AI12-0441-1*} Execution of a program unit can encounter error situations in which normal program execution cannot continue. For example, an arithmetic computation can exceed the maximum allowed value of a number, or an attempt can be made to access an array component by using an incorrect index value. To deal with such error situations, the statements of a program unit can be textually followed by exception handlers that specify the actions to be taken when the error situation arises. Exceptions can be raised explicitly by a raise statement. 29/5

Data Types 30

{*AI12-0313-1*} Every object in the language has a type, which characterizes a set of values and a set of applicable operations. The main categories of types are elementary types (comprising enumeration, numeric, and access types) and composite types (including array and record types). 31/5

32/2 {*AI95-00285-01*} {*AI95-00387-01*} An enumeration type defines an ordered set of distinct enumeration literals, for example a list of states or an alphabet of characters. The enumeration types Boolean, Character, Wide_Character, and Wide_Wide_Character are predefined.

33 Numeric types provide a means of performing exact or approximate numerical computations. Exact computations use integer types, which denote sets of consecutive integers. Approximate computations use either fixed point types, with absolute bounds on the error, or floating point types, with relative bounds on the error. The numeric types Integer, Float, and Duration are predefined.

34/2 {*AI95-00285-01*} {*AI95-00387-01*} Composite types allow definitions of structured objects with related components. The composite types in the language include arrays and records. An array is an object with indexed components of the same type. A record is an object with named components of possibly different types. Task and protected types are also forms of composite types. The array types String, Wide_String, and Wide_Wide_String are predefined.

35/5 {*AI12-0441-1*} Record, task, and protected types can have special components called discriminants which parameterize the type. Variant record structures that depend on the values of discriminants can be defined within a record type.

36/5 {*AI12-0441-1*} Access types allow the construction of linked data structures. A value of an access type represents a reference to an object declared as aliased or to an object created by the evaluation of an allocator. Several variables of an access type can designate the same object, and components of one object can designate the same or other objects. Both the elements in such linked data structures and their relation to other elements can be altered during program execution. Access types also permit references to subprograms to be stored, passed as parameters, and ultimately dereferenced as part of an indirect call.

37 Private types permit restricted views of a type. A private type can be defined in a package so that only the logically necessary properties are made visible to the users of the type. The full structural details that are externally irrelevant are then only available within the package and any child units.

38/5 {*AI12-0438-1*} {*AI12-0441-1*} From any type a new type can be defined by derivation. A type, together with its derivatives (both direct and indirect) form a derivation class. Class-wide operations can be defined that accept as a parameter an operand of any type in a derivation class. For record and private types, the derivatives can be extensions of the parent type. Types that support these object-oriented capabilities of class-wide operations and type extension are tagged, so that the specific type of an operand within a derivation class can be identified at run time. When an operation of a tagged type is applied to an operand whose specific type is not known until run time, implicit dispatching is performed based on the tag of the operand.

38.1/5 {*AI95-00387-01*} {*AI12-0441-1*} Interface types provide abstract models from which other interfaces and types can be composed and derived. This provides a reliable form of multiple inheritance. Interface types can also be implemented by task types and protected types thereby enabling concurrent programming and inheritance to be merged.

39 The concept of a type is further refined by the concept of a subtype, whereby a user can constrain the set of allowed values of a type. Subtypes can be used to define subranges of scalar types, arrays with a limited set of index values, and records and private types with particular discriminant values.

40 *Other Facilities*

41/5 {*AI95-00387-01*} {*AI12-0438-1*} Aspect clauses can be used to specify the mapping between types and features of an underlying machine. For example, the user can specify that objects of a given type are to be represented with a given number of bits, or that the components of a record are to be represented using a given storage layout. Other features allow the controlled use of low level, nonportable, or implementation-dependent aspects, including the direct insertion of machine code.

{*AI12-0313-1*} Aspect clauses can also be used to specify more abstract properties of program entities, such as the pre- and postconditions of a subprogram, or the invariant for a private type. Additional aspects are specifiable to allow user-defined types to use constructs of the language, such as literals, aggregates, or indexing, normally reserved for particular language-defined categories of types, such as numeric types, record types, or array types. 41.1/5

{*AI95-00387-01*} The predefined environment of the language provides for input-output and other capabilities by means of standard library packages. Input-output is supported for values of user-defined as well as of predefined types. Standard means of representing values in display form are also provided. 42/2

{*AI95-00387-01*} The predefined standard library packages provide facilities such as string manipulation, containers of various kinds (vectors, lists, maps, etc.), mathematical functions, random number generation, and access to the execution environment. 42.1/2

{*AI95-00387-01*} The specialized annexes define further predefined library packages and facilities with emphasis on areas such as real-time scheduling, interrupt handling, distributed systems, numerical computation, and high-integrity systems. 42.2/2

Finally, the language provides a powerful means of parameterization of program units, called generic program units. The generic parameters can be types and subprograms (as well as objects and packages) and so allow general algorithms and data structures to be defined that are applicable to all types of a given class. 43

Paragraphs 44 through 57 have been replaced and moved to the Foreword.

Instructions for Comment Submission

{*AI12-0446-1*} Informal comments on this Reference Manual can be submitted in three ways: 57.22/5

- via e-mail to: 57.23/5

  ada-comment@ada-auth.org 57.24/5

- by filling out a web form at: 57.25/5

  https://arg.adaic.org/community-input 57.26/5

- by creating a new "issue" on the ARG GitHub site: 57.27/5

  https://github.com/Ada-Rapporteur-Group/User-Community-Input 57.28/5

{*AI12-0446-1*} If appropriate, the Project Editor will initiate the defect correction procedure. 58/5

Comments should use the following format: 59

> **!topic** *Title summarizing comment* 60/5
> **!reference** Ada 2022 RM*ss.ss(pp)*
> **!from** *Author Name yy-mm-dd*
> **!keywords** *keywords related to topic*
> **!discussion**
>
> *text of discussion*

where *ss.ss* is the clause or subclause number, *pp* is the paragraph number where applicable, and *yy-mm-dd* is the date the comment was sent. The date is optional, as is the **!keywords** line. 61/3

{*AI12-0446-1*} Please use a descriptive "Subject" when sending an e-mail message, or title when creating a new issue, and limit each message to a single comment. 62/5

63 When correcting typographical errors or making minor wording suggestions, please put the correction directly as the topic of the comment; use square brackets [ ] to indicate text to be omitted and curly braces { } to indicate text to be added, and provide enough context to make the nature of the suggestion self-evident or put additional information in the body of the comment, for example:

64 **!topic** [c]{C}haracter
!topic it[']s meaning is not defined

65/5 *This paragraph was deleted.*{*AI12-0446-1*}

Acknowledgements for the Ada 83 edition

65.1/3 Ada is the result of a collective effort to design a common language for programming large scale and real-time systems.

65.2/3 The common high order language program began in 1974. The requirements of the United States Department of Defense were formalized in a series of documents which were extensively reviewed by the Services, industrial organizations, universities, and foreign military departments. The Ada language was designed in accordance with the final (1978) form of these requirements, embodied in the Steelman specification.

65.3/3 The Ada design team was led by Jean D. Ichbiah and has included Bernd Krieg-Brueckner, Brian A. Wichmann, Henry F. Ledgard, Jean-Claude Heliard, Jean-Loup Gailly, Jean-Raymond Abrial, John G.P. Barnes, Mike Woodger, Olivier Roubine, Paul N. Hilfinger, and Robert Firth.

65.4/3 At various stages of the project, several people closely associated with the design team made major contributions. They include J.B. Goodenough, R.F. Brender, M.W. Davis, G. Ferran, K. Lester, L. MacLaren, E. Morel, I.R. Nassi, I.C. Pyle, S.A. Schuman, and S.C. Vestal.

65.5/3 Two parallel efforts that were started in the second phase of this design had a deep influence on the language. One was the development of a formal definition using denotational semantics, with the participation of V. Donzeau-Gouge, G. Kahn, and B. Lang. The other was the design of a test translator with the participation of K. Ripken, P. Boullier, P. Cadiou, J. Holden, J.F. Hueras, R.G. Lange, and D.T. Cornhill. The entire effort benefitted from the dedicated assistance of Lyn Churchill and Marion Myers, and the effective technical support of B. Gravem, W.L. Heimerdinger, and P. Cleve. H.G. Schmitz served as program manager.

65.6/3 Over the five years spent on this project, several intense week-long design reviews were conducted, with the participation of P. Belmont, B. Brosgol, P. Cohen, R. Dewar, A. Evans, G. Fisher, H. Harte, A.L. Hisgen, P. Knueven, M. Kronental, N. Lomuto, E. Ploedereder, G. Seegmueller, V. Stenning, D. Taffs, and also F. Belz, R. Converse, K. Correll, A.N. Habermann, J. Sammet, S. Squires, J. Teller, P. Wegner, and P.R. Wetherall.

65.7/3 Several persons had a constructive influence with their comments, criticisms and suggestions. They include P. Brinch Hansen, G. Goos, C.A.R. Hoare, Mark Rain, W.A. Wulf, and also E. Boebert, P. Bonnard, H. Clausen, M. Cox, G. Dismukes, R. Eachus, T. Froggatt, H. Ganzinger, C. Hewitt, S. Kamin, R. Kotler, O. Lecarme, J.A.N. Lee, J.L. Mansion, F. Minel, T. Phinney, J. Roehrich, V. Schneider, A. Singer, D. Slosberg, I.C. Wand, the reviewers of Ada-Europe, AdaTech, Afcet, those of the LMSC review team, and those of the Ada Tokyo Study Group.

65.8/3 These reviews and comments, the numerous evaluation reports received at the end of the first and second phase, the nine hundred language issue reports and test and evaluation reports received from fifteen different countries during the third phase of the project, the thousands of comments received during the ANSI Canvass, and the on-going work of the IFIP Working Group 2.4 on system implementation

languages and that of the Purdue Europe LTPL-E committee, all had a substantial influence on the final definition of Ada.

The Military Departments and Agencies have provided a broad base of support including funding, extensive reviews, and countless individual contributions by the members of the High Order Language Working Group and other interested personnel. In particular, William A. Whitaker provided leadership for the program during the formative stages. David A. Fisher was responsible for the successful development and refinement of the language requirement documents that led to the Steelman specification. 65.9/3

The Ada 83 language definition was developed by Cii Honeywell Bull and later Alsys, and by Honeywell Systems and Research Center, under contract to the United States Department of Defense. William E. Carlson and later Larry E. Druffel served as the technical representatives of the United States Government and effectively coordinated the efforts of all participants in the Ada program. 65.10/3

Acknowledgements for the Ada 95 edition

This Reference Manual was prepared by the Ada 9X Mapping/Revision Team based at Intermetrics, Inc., which has included: W. Carlson, Program Manager; T. Taft, Technical Director; J. Barnes (consultant); B. Brosgol (consultant); R. Duff (Oak Tree Software); M. Edwards; C. Garrity; R. Hilliard; O. Pazy (consultant); D. Rosenfeld; L. Shafer; W. White; M. Woodger. 66

The following consultants to the Ada 9X Project contributed to the Specialized Needs Annexes: T. Baker (Real-Time/Systems Programming — SEI, FSU); K. Dritz (Numerics — Argonne National Laboratory); A. Gargaro (Distributed Systems — Computer Sciences); J. Goodenough (Real-Time/Systems Programming — SEI); J. McHugh (Secure Systems — consultant); B. Wichmann (Safety-Critical Systems — NPL: UK). 67

This work was regularly reviewed by the Ada 9X Distinguished Reviewers and the members of the Ada 9X Rapporteur Group (XRG): E. Ploedereder, Chairman of DRs and XRG (University of Stuttgart: Germany); B. Bardin (Hughes); J. Barnes (consultant: UK); B. Brett (DEC); B. Brosgol (consultant); R. Brukardt (RR Software); N. Cohen (IBM); R. Dewar (NYU); G. Dismukes (TeleSoft); A. Evans (consultant); A. Gargaro (Computer Sciences); M. Gerhardt (ESL); J. Goodenough (SEI); S. Heilbrunner (University of Salzburg: Austria); P. Hilfinger (UC/Berkeley); B. Källberg (CelsiusTech: Sweden); M. Kamrad II (Unisys); J. van Katwijk (Delft University of Technology: The Netherlands); V. Kaufman (Russia); P. Kruchten (Rational); R. Landwehr (CCI: Germany); C. Lester (Portsmouth Polytechnic: UK); L. Månsson (TELIA Research: Sweden); S. Michell (Multiprocessor Toolsmiths: Canada); M. Mills (US Air Force); D. Pogge (US Navy); K. Power (Boeing); O. Roubine (Verdix: France); A. Strohmeier (Swiss Fed Inst of Technology: Switzerland); W. Taylor (consultant: UK); J. Tokar (Tartan); E. Vasilescu (Grumman); J. Vladik (Prospeks s.r.o.: Czech Republic); S. Van Vlierberghe (OFFIS: Belgium). 68

Other valuable feedback influencing the revision process was provided by the Ada 9X Language Precision Team (Odyssey Research Associates), the Ada 9X User/Implementer Teams (AETECH, Tartan, TeleSoft), the Ada 9X Implementation Analysis Team (New York University) and the Ada community-at-large. 69

Special thanks go to R. Mathis, Convenor of ISO/IEC JTC 1/SC 22 Working Group 9. 70

The Ada 9X Project was sponsored by the Ada Joint Program Office. Christine M. Anderson at the Air Force Phillips Laboratory (Kirtland AFB, NM) was the project manager. 71

Acknowledgements for the Corrigendum version

71.1/3 The editor [R. Brukardt (USA)] would like to thank the many people whose hard work and assistance has made this update possible.

71.2/1 Thanks go out to all of the members of the ISO/IEC JTC 1/SC 22/WG 9 Ada Rapporteur Group, whose work on creating and editing the wording corrections was critical to the entire process. Especially valuable contributions came from the chairman of the ARG, E. Ploedereder (Germany), who kept the process moving; J. Barnes (UK) and K. Ishihata (Japan), whose extremely detailed reviews kept the editor on his toes; G. Dismukes (USA), M. Kamrad (USA), P. Leroy (France), S. Michell (Canada), T. Taft (USA), J. Tokar (USA), and other members too numerous to mention.

71.3/1 Special thanks go to R. Duff (USA) for his explanations of the previous system of formatting of these documents during the tedious conversion to more modern formats. Special thanks also go to the convenor of ISO/IEC JTC 1/SC 22/WG 9, J. Moore (USA), without whose help and support the Corrigendum and this consolidated reference manual would not have been possible.

Acknowledgements for the Amendment 1 version

71.4/3 The editor [R. Brukardt (USA)] would like to thank the many people whose hard work and assistance has made this update possible.

71.5/2 Thanks go out to all of the members of the ISO/IEC JTC 1/SC 22/WG 9 Ada Rapporteur Group, whose work on creating and editing the wording corrections was critical to the entire process. Especially valuable contributions came from the chairman of the ARG, P. Leroy (France), who kept the process on schedule; J. Barnes (UK) whose careful reviews found many typographical errors; T. Taft (USA), who always seemed to have a suggestion when we were stuck, and who also was usually able to provide the valuable service of explaining why things were as they are; S. Baird (USA), who found many obscure problems with the proposals; and A. Burns (UK), who pushed many of the real-time proposals to completion. Other ARG members who contributed were: R. Dewar (USA), G. Dismukes (USA), R. Duff (USA), K. Ishihata (Japan), S. Michell (Canada), E. Ploedereder (Germany), J.P. Rosen (France), E. Schonberg (USA), J. Tokar (USA), and T. Vardanega (Italy).

71.6/2 Special thanks go to Ada-Europe and the Ada Resource Association, without whose help and support the Amendment and this consolidated reference manual would not have been possible. M. Heaney (USA) requires special thanks for his tireless work on the containers packages. Finally, special thanks go to the convenor of ISO/IEC JTC 1/SC 22/WG 9, J. Moore (USA), who guided the document through the standardization process.

Acknowledgements for the Ada 2012 edition

71.7/3 The editor [R. Brukardt (USA)] would like to thank the many people whose hard work and assistance has made this revision possible.

71.8/3 Thanks go out to all of the members of the ISO/IEC JTC 1/SC 22/WG 9 Ada Rapporteur Group, whose work on creating and editing the wording changes was critical to the entire process. Especially valuable contributions came from the chairman of the ARG, E. Schonberg (USA), who guided the work; T. Taft (USA), whose insights broke many logjams, both in design and wording; J. Barnes (UK) whose careful reviews uncovered many editorial errors; S. Baird (USA), who repeatedly found obscure interactions with the proposals that the rest of us missed. Other ARG members who substantially contributed were: A. Burns (UK), J. Cousins (UK), R. Dewar (USA), G. Dismukes (USA), R. Duff (USA), P. Leroy (France), B. Moore (Canada), E. Ploedereder (Germany), J.P. Rosen (France), B. Thomas (USA), and T. Vardanega (Italy).

Special thanks go to Ada-Europe and the Ada Resource Association, without whose help and support this third edition of the Ada Standard would not have been possible. A special mention has to go to A. Beneschan (USA) for his efforts in eliminating sloppiness in our wording. M. Heaney (USA) also deserves a mention for his efforts to improve the containers packages. Finally, special thanks go to the convenor of ISO/IEC JTC 1/SC 22/WG 9, J. Tokar (USA), who guided the document through the standardization process. 71.9/3

Acknowledgements for the Ada 2012 Corrigendum 1 version

The editor [R. Brukardt (USA)] would like to thank the many people whose hard work and assistance has made this update possible. 71.10/4

Thanks go out to all of the members of the ISO/IEC JTC 1/SC 22/WG 9 Ada Rapporteur Group, whose work on creating and editing the wording changes was critical to the entire process. Especially valuable contributions came from the chairman of the ARG, J. Cousins (UK), who guided the work; T. Taft (USA), who seems to have the ability to cut any Gordian knot we encounter in wording; J. Barnes (UK) who continues to be able to find editorial errors invisible to most; S. Baird (USA), who so frequently finds obscure interactions that we now have named such things for him. Other ARG members who substantially contributed were: A. Burns (UK), R. Dewar (USA), G. Dismukes (USA), R. Duff (USA), B. Moore (Canada), E. Ploedereder (Germany), J.P. Rosen (France), E. Schonberg (USA), and T. Vardanega (Italy). 71.11/4

Finally, special thanks go to the convenor of ISO/IEC JTC 1/SC 22/WG 9, J. Tokar (USA), who guided the document through the standardization process. 71.12/4

Acknowledgements for the Ada 2022 version

The editor [R. Brukardt] would like to thank the many people whose hard work and assistance has made this revision possible. 71.13/5

Thanks go out to all of the members of the ISO/IEC JTC 1/SC 22/WG 9 Ada Rapporteur Group, whose work in all steps of the process, from determining problems to address, reviewing feature designs, and creating and editing wording changes, was critical to the entire process. Especially valuable contributions came from the chairman of the ARG through June 2018, J. Cousins, who guided the work and ensured we followed defined procedures; his replacement as chairman, S. Baird, who ably powered through obstacles to complete the work while continuing to find obscure interactions; T. Taft, who often solved difficult problems that had stumped others; B. Moore, whose frequent suggestions for parallel constructs greatly improved the result. Other ARG members who substantially contributed were: R. Amiard, J. Barnes, A. Burns, A. Charlet, G. Dismukes, C. Dross, R. Duff, E. Fish, E. Ploedereder, J.P. Rosen, F. Schanda, E. Schonberg, J. Squirek, T. Vardanega, and R. Wai. 71.14/5

Finally, special thanks go to the convenor of ISO/IEC JTC 1/SC 22/WG 9, P. Rogers, who guided the document through the standardization process. 71.15/5

Using this version of the Ada Reference Manual

This document has been revised with the corrections specified in Technical Corrigendum 1 for Ada 2012 (which corresponds to ISO/IEC 8652:2012/COR.1:2016) and other changes specifically for this fourth edition. In addition, more annotations have been added and a variety of editorial errors have been corrected. 72/5

Changes to the original 1995 version of the Annotated Ada Reference Manual (AARM) can be identified by the version number following the paragraph number. Paragraphs with a version number of /1 were changed by Technical Corrigendum 1 for Ada 95 or were editorial corrections at that time, while 73/5

paragraphs with a version number of /2 were changed by Amendment 1 or were more recent editorial corrections, and paragraphs with a version number of /3 were changed by the 2012 edition of the AARM (including additional editorial corrections). Paragraphs with a version number of /4 were changed by Technical Corrigendum 1 for Ada 2012 or were editorial corrections at that time. Paragraphs with a version number of /5 were changed by the 2022 edition of the AARM (including additional editorial corrections). Paragraphs not so marked are unchanged since the original 1995 edition of the Annotated Ada Reference Manual, and have the same paragraph numbers as in that edition. Inserted text is indicated by underlining, and deleted text is indicated by strikethroughs. Some versions also use color to indicate the version of the change. Where paragraphs are inserted, the paragraph numbers are of the form pp.nn, where pp is the number of the preceding paragraph, and nn is an insertion number. For instance, the first paragraph inserted after paragraph 8 is numbered 8.1, the second paragraph inserted is numbered 8.2, and so on. Deleted paragraphs are indicated by text such as *This paragraph was deleted.* Deleted paragraphs include empty paragraphs that were numbered in the 1995 edition of the Ada Reference Manual. Similar markings and numbering are used for changes to annotations.

73.a/4 **To be honest:** For normative text, the paragraph number is considered part of the paragraph; when a paragraph is moved to a different paragraph number, it is marked as changed even if the contents have not changed. For non-normative text such as NOTES and Examples, paragraph numbers of unchanged context may change without indication. This may happen if additional normative content is added to a subclause.

1 General

Discussion: This Annotated Ada Reference Manual (AARM) contains the entire text of the fourth edition of the Ada Reference Manual (referred to here as the Ada 2022 RM), plus certain annotations. The annotations give a more in-depth analysis of the language. They describe the reason for each nonobvious rule, and point out interesting ramifications of the rules and interactions among the rules (interesting to language lawyers, that is). Differences between Ada 83, Ada 95, Ada 2005, Ada 2012, and Ada 2022 are listed. (The text you are reading now is an annotation.) 2.a/5

The AARM stresses detailed correctness and uniformity over readability and understandability. We're not trying to make the language "appear" simple here; on the contrary, we're trying to expose hidden complexities, so we can more easily detect language bugs. The Ada 2022 RM, on the other hand, is intended to be a more readable document for programmers. 2.b/3

The annotations in the AARM are as follows: 2.c

- Text that is logically redundant is shown [in square brackets, like this]. Technically, such text could be written as a Note in the Ada 2022 RM (and the Ada 95 and 2005 RMs before it), since it is really a theorem that can be proven from the nonredundant rules of the language. We use the square brackets instead when it seems to make the Ada 2022 RM more readable. 2.d/3
- The rules of the language (and some AARM-only text) are categorized, and placed under certain *sub-headings* that indicate the category. For example, the distinction between Name Resolution Rules and Legality Rules is particularly important, as explained in 8.6. 2.e
- Text under the following sub-headings appears in both documents: 2.f
  - The unlabeled text at the beginning of each clause or subclause, 2.g
  - Syntax, 2.h
  - Name Resolution Rules, 2.i
  - Legality Rules, 2.j
  - Static Semantics, 2.k
  - Post-Compilation Rules, 2.l
  - Dynamic Semantics, 2.m
  - Bounded (Run-Time) Errors, 2.n
  - Erroneous Execution, 2.o
  - Implementation Requirements, 2.p
  - Documentation Requirements, 2.q
  - Metrics, 2.r
  - Implementation Permissions, 2.s
  - Implementation Advice, 2.t
  - NOTES, 2.u
  - Examples. 2.v
- Text under the following sub-headings does not appear in the Ada 2022 RM: 2.w/3
  - Language Design Principles, 2.x
  - Inconsistencies With Ada 83, 2.y
  - Incompatibilities With Ada 83, 2.z
  - Extensions to Ada 83, 2.aa
  - Wording Changes from Ada 83, 2.bb/2
  - Inconsistencies With Ada 95, 2.bb.1/2
  - Incompatibilities With Ada 95, 2.bb.2/2
  - Extensions to Ada 95, 2.bb.3/2
  - Wording Changes from Ada 95, 2.bb.4/3
  - Inconsistencies With Ada 2005, 2.bb.5/3
  - Incompatibilities With Ada 2005, 2.bb.6/3
  - Extensions to Ada 2005, 2.bb.7/3

2.bb.8/4 • Wording Changes from Ada 2005,

2.bb.9/4 • Inconsistencies With Ada 2012,

2.bb.10/4 • Incompatibilities With Ada 2012,

2.bb.11/4 • Extensions to Ada 2012,

2.bb.12/4 • Wording Changes from Ada 2012.

2.cc • The AARM also includes the following kinds of annotations. These do not necessarily annotate the immediately preceding rule, although they often do.

2.dd **Reason:** An explanation of why a certain rule is necessary, or why it is worded in a certain way.

2.ee **Ramification:** An obscure ramification of the rules that is of interest only to language lawyers. (If a ramification of the rules is of interest to programmers, then it appears under NOTES.)

2.ff **Proof:** An informal proof explaining how a given Note or [marked-as-redundant] piece of text follows from the other rules of the language.

2.gg **Implementation Note:** A hint about how to implement a feature, or a particular potential pitfall that an implementer needs to be aware of.

2.hh **Change:** Change annotations are not used in this version. Changes from previous versions have been removed. Changes in this version are marked with versioned paragraph numbers, as explained in the "Corrigendum Changes" clause of the "Introduction".

2.ii **Discussion:** Other annotations not covered by the above.

2.jj **To be honest:** A rule that is considered logically necessary to the definition of the language, but which is so obscure or pedantic that only a language lawyer would care. These are the only annotations that could be considered part of the language definition.

2.jj.1/5 **Term entry:** {*AI12-0443-1*} The text of a Term and Definition entry — this text will also appear in 1.3, "Terms and Definitions".

2.kk/3 **Discussion:** In general, the Ada 2022 RM text appears in the normal font, whereas AARM-only text appears in a smaller font. Notes also appear in the smaller font, as recommended by ISO/IEC style guidelines. Ada examples are also usually printed in a smaller font.

2.ll If you have trouble finding things, be sure to use the index. Each defined term appears there, and also in *italics, like this*. Syntactic categories defined in BNF are also indexed.

2.mm A definition marked "[distributed]" is the main definition for a term whose complete definition is given in pieces distributed throughout the document. The pieces are marked "[partial]" or with a phrase explaining what cases the partial definition applies to.

1.1 Scope

1/3 {*AI05-0299-1*} This Reference Manual specifies the form and meaning of programs written in Ada. Its purpose is to promote the portability of Ada programs to a variety of computing systems.

1.1.1 Extent

1 This Reference Manual specifies:

2 • The form of a program written in Ada;

3 • The effect of translating and executing such a program;

4/5 • The manner in which program units can be combined to form Ada programs;

5 • The language-defined library units that a conforming implementation is required to supply;

6/5 • {*AI12-0441-1*} The permissible variations in conformance to the rules of this document, and the manner in which they are to be documented;

7/5 • {*AI12-0441-1*} Those violations of the requirements of this document that a conforming implementation is required to detect, and the effect of attempting to translate or execute a program containing such violations;

- {*AI12-0441-1*} Those violations of the requirements of this document that a conforming implementation is not required to detect. 8/5

This Reference Manual does not specify: 9

- The means whereby a program written in Ada is transformed into object code executable by a processor; 10
- The means whereby translation or execution of programs is invoked and the executing units are controlled; 11
- The size or speed of the object code, or the relative execution speed of different language constructs; 12
- The form or contents of any listings produced by implementations; in particular, the form or contents of error or warning messages; 13
- {*AI12-0425-1*} The effect of unspecified execution; 14/5
- The size of a program or program unit that will exceed the capacity of a particular conforming implementation. 15

1.1.2 Structure

{*AI05-0299-1*} This document contains thirteen clauses, fifteen annexes, and an index. 1/3

Discussion: {*AI05-0299-1*} What Ada 83 called a “chapter” and Ada 95 (and Ada 2005) called a “section” is called a “clause” in this Reference Manual. Similarly, what Ada 83 called a “section” and Ada 95 (and Ada 2005) called a “clause” is called a “subclause” in this Reference Manual. Confused yet? This terminology is out of our hands; it is (and was) forced by ever-changing ISO rules for drafting Reference Manuals. 1.a/3

The *core* of the Ada language consists of: 2

- {*AI05-0299-1*} Clauses 1 through 13 3/3
- Annex A, “Predefined Language Environment” 4
- Annex B, “Interface to Other Languages” 5
- Annex J, “Obsolescent Features” 6

Term entry: core language — clause or annex in which are defined language constructs or capabilities that are provided by all conforming implementations 6.a/5
Note: A construct is said to be part of the core language if it is defined in a core language clause or annex.

The following *Specialized Needs Annexes* define features that are needed by certain application areas: 7

- Annex C, “Systems Programming” 8
- Annex D, “Real-Time Systems” 9
- Annex E, “Distributed Systems” 10
- Annex F, “Information Systems” 11
- Annex G, “Numerics” 12
- Annex H, “High Integrity Systems” 13

Term entry: specialized needs annex — annex in which are defined language constructs or capabilities that are not necessarily provided by all conforming implementations 13.a/5

The core language and the Specialized Needs Annexes are normative, except that the material in each of the items listed below is informative: 14

- Text under a NOTES or Examples heading. 15

16/3 • {*AI05-0299-1*} Each subclause whose title starts with the word “Example” or “Examples”.

17 All implementations shall conform to the core language. In addition, an implementation may conform separately to one or more Specialized Needs Annexes.

18 The following Annexes are informative:

19 • Annex K, “Language-Defined Aspects and Attributes”

20 • Annex L, “Language-Defined Pragmas”

21/3 • {*AI05-0004-1*} Annex M, “Summary of Documentation Requirements”

22 • Annex N, “Glossary”

23 • Annex P, “Syntax Summary”

23.1/3 • {*AI05-0262-1*} Annex Q, “Language-Defined Entities”

23.a **Discussion:** The idea of the Specialized Needs Annexes is that implementations can choose to target certain application areas. For example, an implementation specifically targeted to embedded machines might support the application-specific features for Real-time Systems, but not the application-specific features for Information Systems.

23.b The Specialized Needs Annexes extend the core language only in ways that users, implementations, and standards bodies are allowed to extend the language; for example, via additional library units, attributes, representation items (see 13.1), pragmas, and constraints on semantic details that are left unspecified by the core language. Many implementations already provide much of the functionality defined by Specialized Needs Annexes; our goal is to increase uniformity among implementations by defining standard ways of providing the functionality.

23.c/2 {*AI95-00114-01*} We recommend that the certification procedures allow implementations to certify the core language, plus any set of the Specialized Needs Annexes. We recommend that implementations *not* be allowed to certify a portion of one of the Specialized Needs Annexes, although implementations can, of course, provide uncertified support for such portions. We have designed the Specialized Needs Annexes assuming that this recommendation is followed. Thus, our decisions about what to include and what not to include in those annexes are based on the assumption that each annex is certified in an “all-or-nothing” manner.

23.d An implementation may, of course, support extensions that are different from (but possibly related to) those defined by one of the Specialized Needs Annexes. We recommend that, where appropriate, implementations do this by adding library units that are children of existing language-defined library packages.

23.e An implementation should not provide extensions that conflict with those defined in the Specialized Needs Annexes, in the following sense: Suppose an implementation supports a certain error-free program that uses only functionality defined in the core and in the Specialized Needs Annexes. The implementation should ensure that that program will still be error free in some possible full implementation of all of the Specialized Needs Annexes, and that the semantics of the program will not change. For example, an implementation should not provide a package with the same name as one defined in one of the Specialized Needs Annexes, but that behaves differently, *even if that implementation does not claim conformance to that Annex*.

23.f Note that the Specialized Needs Annexes do not conflict with each other; it is the intent that a single implementation can conform to all of them.

24/4 {*AI05-0299-1*} {*AI12-0056-1*} Each clause is divided into subclauses that have a common structure. Each clause and subclause first introduces its subject. After the introductory text, text is labeled with the following headings:

Language Design Principles

24.a These are not rules of the language, but guiding principles or goals used in defining the rules of the language. In some cases, the goal is only partially met; such cases are explained.

24.b/3 {*AI05-0005-1*} This is not part of the definition of the language, and does not appear in the Ada 2022 RM.

Syntax

25 Syntax rules (indented).

Name Resolution Rules

{*AI05-0299-1*} Compile-time rules that are used in name resolution, including overload resolution. 26/3

Discussion: These rules are observed at compile time. (We say "observed" rather than "checked", because these rules are not individually checked. They are really just part of the Legality Rules in Clause 8 that require exactly one interpretation of each constituent of a complete context.) The only rules used in overload resolution are the Syntax Rules and the Name Resolution Rules. 26.a

When dealing with nonoverloadable declarations it sometimes makes no semantic difference whether a given rule is a Name Resolution Rule or a Legality Rule, and it is sometimes difficult to decide which it should be. We generally make a given rule a Name Resolution Rule only if it has to be. For example, "The name, if any, in a raise_statement shall be the name of an exception." is under "Legality Rules." 26.b

Legality Rules

Rules that are enforced at compile time. A construct is *legal* if it obeys all of the Legality Rules. 27

Discussion: These rules are not used in overload resolution. 27.a

Note that run-time errors are always attached to exceptions; for example, it is not "illegal" to divide by zero, it just raises an exception. 27.b

Static Semantics

A definition of the compile-time effect of each construct. 28

Discussion: The most important compile-time effects represent the effects on the symbol table associated with declarations (implicit or explicit). In addition, we use this heading as a bit of a grab bag for equivalences, package specifications, etc. For example, this is where we put statements like so-and-so is equivalent to such-and-such. (We ought to try to really mean it when we say such things!) Similarly, statements about magically-generated implicit declarations go here. These rules are generally written as statements of fact about the semantics, rather than as a you-shall-do-such-and-such sort of thing. 28.a

Post-Compilation Rules

Rules that are enforced before running a partition. A partition is legal if its compilation units are legal and it obeys all of the Post-Compilation Rules. 29

Discussion: It is not specified exactly when these rules are checked, so long as they are checked for any given partition before that partition starts running. An implementation may choose to check some such rules at compile time, and reject compilation_units accordingly. Alternatively, an implementation may check such rules when the partition is created (usually known as "link time"), or when the partition is mapped to a particular piece of hardware (but before the partition starts running). 29.a

Dynamic Semantics

A definition of the run-time effect of each construct. 30

Discussion: This heading describes what happens at run time. runtime checks, which raise exceptions upon failure, are described here. Each item that involves a runtime check is marked with the name of the check — these are the same check names that are used in a pragma Suppress. Principle: Every check should have a name, usable in a pragma Suppress. 30.a

Bounded (Run-Time) Errors

Situations that result in bounded (run-time) errors (see 1.1.5). 31

Discussion: The "bounds" of each such error are described here — that is, we characterize the set of all possible behaviors that can result from a bounded error occurring at run time. 31.a

Erroneous Execution

Situations that result in erroneous execution (see 1.1.5). 32

Implementation Requirements

Additional requirements for conforming implementations. 33

33.a **Discussion:** ...as opposed to rules imposed on the programmer. An example might be, “The smallest representable duration, Duration'Small, shall not be greater than twenty milliseconds”.

33.b It's really just an issue of how the rule is worded. We could write the same rule as “The smallest representable duration is an implementation-defined value less than or equal to 20 milliseconds” and then it would be under “Static Semantics”.

Documentation Requirements

34 Documentation requirements for conforming implementations.

34.a **Discussion:** These requirements are beyond those that are implicitly specified by the phrase “implementation defined”. The latter require documentation as well, but we don't repeat these cases under this heading. Usually this heading is used for when the description of the documentation requirement is longer and does not correspond directly to one, narrow normative sentence.

Metrics

35 Metrics that are specified for the time/space properties of the execution of certain language constructs.

Implementation Permissions

36 Additional permissions given to the implementer.

36.a **Discussion:** For example, “The implementation is allowed to impose further restrictions on the record aggregates allowed in code statements.” When there are restrictions on the permission, those restrictions are given here also. For example, “An implementation is allowed to restrict the kinds of subprograms that are allowed to be main subprograms. However, it shall support at least parameterless procedures.” — we don't split this up between here and “Implementation Requirements”.

Implementation Advice

37 Optional advice given to the implementer. The word “should” is used to indicate that the advice is a recommendation, not a requirement. It is implementation defined whether or not a given recommendation is obeyed.

37.a/2 **Implementation defined:** Whether or not each recommendation given in Implementation Advice is followed — see M.3 for a listing.

37.b/1 **Discussion:** The advice generally shows the intended implementation, but the implementer is free to ignore it. The implementer is the sole arbiter of whether or not the advice has been obeyed, if not, whether the reason is a good one, and whether the required documentation is sufficient. It would be wrong for the ACATS to enforce any of this advice.

37.c For example, “Whenever possible, the implementation should choose a value no greater than fifty microseconds for the smallest representable duration, Duration'Small.”

37.d We use this heading, for example, when the rule is so low level or implementation-oriented as to be untestable. We also use this heading when we wish to encourage implementations to behave in a certain way in most cases, but we do not wish to burden implementations by requiring the behavior.

38 NOTE Notes emphasize consequences of the rules described in the (sub)clause or elsewhere. This material is informative.

Examples

39 Examples illustrate the possible forms of the constructs described. This material is informative.

39.a.1/5 **Discussion:** {*AI12-0386-1*} Names used in examples refer either to language-defined entities or to entities declared in other Examples in this Reference Manual. In the latter case, if the reference is to a later example, we use a cross-reference (-- *See <<some clause>>*.) to show the source of the declaration. In theory, one could combine all of the examples (excepting examples of fragments like identifiers or type_definitions) in an appropriate unit and compile them all.

39.a

The next three headings list all language changes between Ada 83 and Ada 95. Language changes are any change that changes the set of text strings that are legal Ada programs, or changes the meaning of any legal program. Wording changes, such as changes in terminology, are not language changes. Each language change falls into one of the following three categories:

Inconsistencies With Ada 83

This heading lists all of the upward inconsistencies between Ada 83 and Ada 95. Upward inconsistencies are situations in which a legal Ada 83 program is a legal Ada 95 program with different semantics. This type of upward incompatibility is the worst type for users, so we only tolerate it in rare situations. 39.b

(Note that the semantics of a program is not the same thing as the behavior of the program. Because of Ada's indeterminacy, the "semantics" of a given feature describes a *set* of behaviors that can be exhibited by that feature. The set can contain more than one allowed behavior. Thus, when we ask whether the semantics changes, we are asking whether the set of behaviors changes.) 39.c

This is not part of the definition of the language, and does not appear in the Ada 95, Ada 2005, Ada 2012, or Ada 2022 RM. 39.d/4

Incompatibilities With Ada 83

This heading lists all of the upward incompatibilities between Ada 83 and Ada 95, except for the ones listed under "Inconsistencies With Ada 83" above. These are the situations in which a legal Ada 83 program is illegal in Ada 95. We do not generally consider a change that turns erroneous execution into an exception, or into an illegality, to be upwardly incompatible. 39.e

This is not part of the definition of the language, and does not appear in the Ada 95, Ada 2005, Ada 2012, or Ada 2022 RM. 39.f/4

Extensions to Ada 83

This heading is used to list all upward compatible language changes; that is, language extensions. These are the situations in which a legal Ada 95 program is not a legal Ada 83 program. The vast majority of language changes fall into this category. 39.g

This is not part of the definition of the language, and does not appear in the Ada 95, Ada 2005, Ada 2012, or Ada 2022 RM. 39.h/4

39.i

As explained above, the next heading does not represent any language change:

Wording Changes from Ada 83

This heading lists some of the nonsemantic changes between the Ada 83 RM and the Ada 95 RM. It is incomplete; we have not attempted to list all wording changes, but only the "interesting" ones. 39.j/2

This is not part of the definition of the language, and does not appear in the Ada 95, Ada 2005, Ada 2012, or Ada 2022 RM. 39.k/4

Discussion: 39.l/2

The next three headings list all language changes between Ada 95 and Ada 2005 (the language defined by the Ada 95 standard plus Technical Corrigendum 1 plus Amendment 1). Each language change falls into one of the following three categories:

Inconsistencies With Ada 95

This heading lists all of the upward inconsistencies between Ada 95 and Ada 2005. Upward inconsistencies are situations in which a legal Ada 95 program is a legal Ada 2005 program with different semantics. 39.m/2

{*AI05-0005-1*} Inconsistencies marked with **Corrigendum:** are corrections to the original Ada 95 definition introduced by Corrigendum 1. Inconsistencies marked with **Amendment Correction:** are corrections to the original Ada 95 definition added by Amendment 1. Formally, these are inconsistencies caused by Ada Issues classified as Binding Interpretations; implementations of Ada 95 are supposed to follow these corrections, not the original flawed language definition. Thus, these strictly speaking are not inconsistencies between Ada 95 and Ada 2005. Practically, however, they very well may be, as early Ada 95 implementations might not follow the recommendation. Inconsistencies so marked are not portable between Ada 95 implementations, while usually Ada 2005 will have more clearly defined behavior. Therefore, we document these for completeness. 39.n/3

This is not part of the definition of the language, and does not appear in the Ada 2005 , Ada 2012, or Ada 2022 RM. 39.o/4

Incompatibilities With Ada 95

This heading lists all of the upward incompatibilities between Ada 95 and Ada 2005, except for the ones listed under "Inconsistencies With Ada 95" above. These are the situations in which a legal Ada 95 program is illegal in Ada 2005. 39.p/2

39.q/3 {*AI05-0005-1*} As with inconsistencies, incompatibilities marked with **Corrigendum:** are corrections to the original Ada 95 definition introduced by Corrigendum 1. Incompatibilities marked with **Amendment Correction:** are corrections to the original Ada 95 definition added by Amendment 1. Formally, these are incompatibilities caused by Ada Issues classified as Binding Interpretations; implementations of Ada 95 are supposed to follow these corrections, not the original flawed language definition. Thus, these strictly speaking are not incompatibilities between Ada 95 and Ada 2005. Practically, however, they very well may be, as early Ada 95 implementations might not follow the recommendation. Therefore, some Ada 95 implementations may be able to compile the examples, while others might not. In contrast, Ada 2005 compilers will have consistent behavior. Therefore, we document these for completeness.

39.r/4 This is not part of the definition of the language, and does not appear in the Ada 2005 , Ada 2012, or Ada 2022 RM.

Extensions to Ada 95

39.s/2 This heading is used to list all upward compatible language changes; that is, language extensions. These are the situations in which a legal Ada 2005 program is not a legal Ada 95 program. The vast majority of language changes fall into this category.

39.t/3 {*AI05-0005-1*} As with incompatibilities, extensions marked with **Corrigendum:** are corrections to the original Ada 95 definition introduced by Corrigendum 1. Extensions marked with **Amendment Correction:** are corrections to the original Ada 95 definition added by Amendment 1. Formally, these are extensions allowed by Ada Issues classified as Binding Interpretations. As corrections, implementations of Ada 95 are allowed to implement these extensions. Thus, these strictly speaking are not extensions of Ada 95; they're part of Ada 95. Practically, however, they very well may be extensions, as early Ada 95 implementations might not implement the extension. Therefore, some Ada 95 implementations may be able to compile the examples, while others might not. In contrast, Ada 2005 compilers will always support the extensions. Therefore, we document these for completeness.

39.u/4 This is not part of the definition of the language, and does not appear in the Ada 2005 , Ada 2012, or Ada 2022 RM.

39.v/2

As explained above, the next heading does not represent any language change:

Wording Changes from Ada 95

39.w/2 This heading lists some of the nonsemantic changes between the Ada 95 RM and the Ada 2005 RM. This heading lists only “interesting” changes (for instance, editorial corrections are not listed). Changes which come from Technical Corrigendum 1 are marked **Corrigendum**; unmarked changes come from Amendment 1.

39.x/4 This is not part of the definition of the language, and does not appear in the Ada 2005 , Ada 2012, or Ada 2022 RM.

39.y/3 **Discussion:**

The next three headings list all language changes between Ada 2005 (the language defined by the Ada 95 standard plus Technical Corrigendum 1 plus Amendment 1) and Ada 2012 (the language defined by the third edition of the Reference Manual). Each language change falls into one of the following three categories:

Inconsistencies With Ada 2005

39.z/3 This heading lists all of the upward inconsistencies between Ada 2005 and Ada 2012. Upward inconsistencies are situations in which a legal Ada 2005 program is a legal Ada 2012 program with different semantics.

39.aa/3 Inconsistencies marked with **Correction:** are corrections to the original Ada 2005 definition added by the third edition of the Reference Manual. Formally, these are inconsistencies caused by Ada Issues classified as Binding Interpretations; implementations of Ada 2005 are supposed to follow these corrections, not the original flawed language definition. Thus, these strictly speaking are not inconsistencies between Ada 2005 and Ada 2012. Practically, however, they very well may be, as early Ada 2005 implementations might not follow the recommendation. Inconsistencies so marked are not portable between Ada 2005 implementations, while usually Ada 2012 will have more clearly defined behavior. Therefore, we document these for completeness.

39.bb/4 This is not part of the definition of the language, and does not appear in the Ada 2012 or Ada 2022 RM.

Incompatibilities With Ada 2005

39.cc/3 This heading lists all of the upward incompatibilities between Ada 2005 and Ada 2012, except for the ones listed under “Inconsistencies With Ada 2005” above. These are the situations in which a legal Ada 2005 program is illegal in Ada 2012.

39.dd/3 As with inconsistencies, incompatibilities marked with **Correction:** are corrections to the original Ada 2005 definition added by the third edition. Formally, these are incompatibilities caused by Ada Issues classified as Binding Interpretations; implementations of Ada 2005 are supposed to follow these corrections, not the original flawed language definition. Thus, these strictly speaking are not incompatibilities between Ada 2005 and Ada 2012.

Practically, however, they very well may be, as early Ada 2005 implementations might not follow the recommendation. Therefore, some Ada 2005 implementations may be able to compile the examples, while others might not. In contrast, Ada 2012 compilers will have consistent behavior. Therefore, we document these for completeness.

This is not part of the definition of the language, and does not appear in the Ada 2012 or Ada 2022 RM. 39.ee/4

Extensions to Ada 2005

This heading is used to list all upward compatible language changes; that is, language extensions. These are the situations in which a legal Ada 2012 program is not a legal Ada 2005 program. The vast majority of language changes fall into this category. 39.ff/3

As with incompatibilities, extensions marked with **Correction:** are corrections to the original Ada 2005 definition added by the third edition. Formally, these are extensions allowed by Ada Issues classified as Binding Interpretations. As corrections, implementations of Ada 2005 (and sometimes Ada 95) are allowed to implement these extensions. Thus, these strictly speaking are not extensions of Ada 2005; they're part of Ada 2005. Practically, however, they very well may be extensions, as early Ada 2005 implementations might not implement the extension. Therefore, some Ada 2005 implementations may be able to compile the examples, while others might not. In contrast, Ada 2012 compilers will always support the extensions. Therefore, we document these for completeness. 39.gg/3

This is not part of the definition of the language, and does not appear in the Ada 2012 or Ada 2022 RM. 39.hh/4

39.ii/3

As explained above, the next heading does not represent any language change:

Wording Changes from Ada 2005

This heading lists some of the nonsemantic changes between the Ada 2005 RM and the Ada 2012 RM. This heading lists only “interesting” changes (for instance, editorial corrections are not listed). Items marked **Correction:** come from Ada Issues classified as Binding Interpretations and strictly speaking belong to Ada 2005; other items only belong to Ada 2012. 39.jj/3

This is not part of the definition of the language, and does not appear in the Ada 2012 or Ada 2022 RM. 39.kk/4

Discussion: 39.ll/5

The next three headings list all language changes between Ada 2012 (the language defined by ISO/IEC 8652:2012(E)) and Ada 2022 (the language defined by this edition of the Reference Manual). Each language change falls into one of the following three categories:

Inconsistencies With Ada 2012

This heading lists all of the upward inconsistencies between Ada 2012 and Ada 2022. Upward inconsistencies are situations in which a legal Ada 2012 program is a legal Ada 2022 program with different semantics. 39.mm/5

Inconsistencies marked with **Corrigendum:** are corrections to the original Ada 2012 definition added by Technical Corrigendum 1 to Ada 2012. Inconsistencies marked with **Correction:** are corrections to the original Ada 2012 definition added by this edition of the Reference Manual. Formally, these are inconsistencies caused by Ada Issues classified as Binding Interpretations; implementations of Ada 2012 are supposed to follow these corrections, not the original flawed language definition. Thus, these strictly speaking are not inconsistencies between Ada 2012 and Ada 2022. Practically, however, they very well may be, as early Ada 2012 implementations might not follow the recommendation. Inconsistencies so marked are not portable between Ada 2012 implementations, while usually Ada 2022 will have more clearly defined behavior. Therefore, we document these for completeness. 39.nn/5

This is not part of the definition of the language, and does not appear in any version of the Ada RM. 39.oo/4

Incompatibilities With Ada 2012

This heading lists all of the upward incompatibilities between Ada 2012 and Ada 2022, except for the ones listed under “Inconsistencies With Ada 2012” above. These are the situations in which a legal Ada 2012 program is illegal in Ada 2022. 39.pp/5

As with inconsistencies, incompatibilities marked with **Corrigendum:** are corrections to the original Ada 2012 definition added by Corrigendum 1 to Ada 2012, and incompatibilities marked with **Correction:** are corrections to the original Ada 2012 definition added by this edition. Formally, these are incompatibilities caused by Ada Issues classified as Binding Interpretations; implementations of Ada 2012 are supposed to follow these corrections, not the original flawed language definition. Thus, these strictly speaking are not incompatibilities between Ada 2012 and Ada 2022. Practically, however, they very well may be, as early Ada 2012 implementations might not follow the recommendation. Therefore, some Ada 2012 implementations may be able to compile the examples, while others 39.qq/5

might not. In contrast, Ada 2022 compilers will have consistent behavior. Therefore, we document these for completeness.

39.rr/4 This is not part of the definition of the language, and does not appear in any version of the Ada RM.

Extensions to Ada 2012

39.ss/5 This heading is used to list all upward compatible language changes; that is, language extensions. These are the situations in which a legal Ada 2022 program is not a legal Ada 2012 program. The vast majority of language changes fall into this category.

39.tt/5 As with incompatibilities, extensions marked with **Corrigendum:** are corrections to the original Ada 2012 definition added by Corrigendum 1 to Ada 2012, and extensions marked with **Correction:** are corrections to the original Ada 2012 definition added by this edition. Formally, these are extensions allowed by Ada Issues classified as Binding Interpretations. As corrections, implementations of Ada 2012 (and sometimes Ada 95) are allowed to implement these extensions. Thus, these strictly speaking are not extensions of Ada 2012; they're part of Ada 2012. Practically, however, they very well may be extensions, as early Ada 2012 implementations might not implement the extension. Therefore, some Ada 2012 implementations may be able to compile the examples, while others might not. In contrast, Ada 2022 compilers will always support the extensions. Therefore, we document these for completeness.

39.uu/4 This is not part of the definition of the language, and does not appear in any version of the Ada RM.

39.vv/4

As explained above, the next heading does not represent any language change:

Wording Changes from Ada 2012

39.ww/5 This heading lists some of the nonsemantic changes between the Ada 2012 RM and the Ada 2022 RM. This heading lists only “interesting” changes (for instance, editorial corrections are not listed). Items marked **Corrigendum:** are included in Corrigendum 1 to Ada 2012, and items marked with **Correction:** are added by this edition. In both cases, the items come from Ada Issues classified as Binding Interpretations and strictly speaking belong to Ada 2012; other items only belong to Ada 2022.

39.xx/4 This is not part of the definition of the language, and does not appear in any version of the Ada RM.

1.1.3 Conformity of an Implementation

Implementation Requirements

1 A conforming implementation shall:

1.a **Discussion:** The *implementation* is the software and hardware that implements the language. This includes compiler, linker, operating system, hardware, etc.

1.b We first define what it means to “conform” in general — basically, the implementation has to properly implement the normative rules given throughout the Reference Manual. Then we define what it means to conform to a Specialized Needs Annex — the implementation must support the core features plus the features of that Annex. Finally, we define what it means to “conform to the Reference Manual” — this requires support for the core language, and allows partial (but not conflicting) support for the Specialized Needs Annexes.

2 • Translate and correctly execute legal programs written in Ada, provided that they are not so large as to exceed the capacity of the implementation;

3 • Identify all programs or program units that are so large as to exceed the capacity of the implementation (or raise an appropriate exception at run time);

3.a **Implementation defined:** Capacity limitations of the implementation.

4 • Identify all programs or program units that contain errors whose detection is required by this Reference Manual;

4.a **Discussion:** Note that we no longer use the term “rejection” of programs or program units. We require that programs or program units with errors or that exceed some capacity limit be “identified”. The way in which errors or capacity problems are reported is not specified.

4.b An implementation is allowed to use standard error-recovery techniques. We do not disallow such techniques from being used across compilation_unit or compilation boundaries.

4.c See also the Implementation Requirements of 10.2, which disallow the execution of illegal partitions.

- Supply all language-defined library units required by this Reference Manual; 5

  Implementation Note: An implementation cannot add to or modify the visible part of a language-defined library unit, except where such permission is explicitly granted, unless such modifications are semantically neutral with respect to the client compilation units of the library unit. An implementation defines the contents of the private part and body of language-defined library units. 5.a

  An implementation can add with_clauses and use_clauses, since these modifications are semantically neutral to clients. (The implementation might need with_clauses in order to implement the private part, for example.) Similarly, an implementation can add a private part even in cases where a private part is not shown in the Reference Manual. Explicit declarations can be provided implicitly or by renaming, provided the changes are semantically neutral. 5.b

  Wherever in the Reference Manual the text of a language-defined library unit contains an italicized phrase starting with "*implementation-defined*", the implementation's version will replace that phrase with some implementation-defined text that is syntactically legal at that place, and follows any other applicable rules. 5.c

  Note that modifications are permitted, even if there are other tools in the environment that can detect the changes (such as a program library browser), so long as the modifications make no difference with respect to the static or dynamic semantics of the resulting programs, as defined by the Reference Manual. 5.d

- Contain no variations except those explicitly permitted by this Reference Manual, or those that are impossible or impractical to avoid given the implementation's execution environment; 6

  Implementation defined: Variations from the standard that are impractical to avoid given the implementation's execution environment. 6.a

  Reason: The "impossible or impractical" wording comes from AI-325. It takes some judgement and common sense to interpret this. Restricting compilation units to less than 4 lines is probably unreasonable, whereas restricting them to less than 4 billion lines is probably reasonable (at least given today's technology). We do not know exactly where to draw the line, so we have to make the rule vague. 6.b

- Specify all such variations in the manner prescribed by this Reference Manual. 7

The *external effect* of the execution of an Ada program is defined in terms of its interactions with its external environment. The following are defined as *external interactions*: 8

- Any interaction with an external file (see A.7); 9
- The execution of certain code_statements (see 13.8); which code_statements cause external interactions is implementation defined. 10

  Implementation defined: Which code_statements cause external interactions. 10.a

- Any call on an imported subprogram (see Annex B), including any parameters passed to it; 11
- Any result returned or exception propagated from a main subprogram (see 10.2) or an exported subprogram (see Annex B) to an external caller; 12

  Discussion: By "result returned" we mean to include function results and values returned in [**in**] **out** parameters. 12.a

  {*8652/0094*} {*AI95-00119-01*} The lack of a result from a program that does not terminate is also included here. 12.a.1/1

- [Any read or update of an atomic or volatile object (see C.6);] 13
- The values of imported and exported objects (see Annex B) at the time of any other interaction with the external environment. 14

  To be honest: {*AI05-0229-1*} Also other uses of imported and exported entities, as defined by the implementation, if the implementation supports such importing or exporting. 14.a/3

A conforming implementation of this Reference Manual shall produce for the execution of a given Ada program a set of interactions with the external environment whose order and timing are consistent with the definitions and requirements of this Reference Manual for the semantics of the given program. 15

Ramification: There is no need to produce any of the "internal effects" defined for the semantics of the program — all of these can be optimized away — so long as an appropriate sequence of external interactions is produced. 15.a

Discussion: See also 11.6 which specifies various liberties associated with optimizations in the presence of language-defined checks, that could change the external effects that might be produced. These alternative external effects are still consistent with the standard, since 11.6 is part of the standard. 15.b

15.c Note also that we only require "*an appropriate* sequence of external interactions" rather than "*the same* sequence..." An optimizer may cause a different sequence of external interactions to be produced than would be produced without the optimizer, so long as the new sequence still satisfies the requirements of the standard. For example, optimization might affect the relative rate of progress of two concurrent tasks, thereby altering the order in which two external interactions occur.

15.d/2 Note that the Ada 83 RM explicitly mentions the case of an "exact effect" of a program, but since so few programs have their effects defined that exactly, we don't even mention this "special" case. In particular, almost any program that uses floating point or tasking has to have some level of inexactness in the specification of its effects. And if one includes aspects of the timing of the external interactions in the external effect of the program (as is appropriate for a real-time language), no "exact effect" can be specified. For example, if two external interactions initiated by a single task are separated by a "**delay** 1.0;" then the language rules imply that the two external interactions have to be separated in time by at least one second, as defined by the clock associated with the delay_relative_statement. This in turn implies that the time at which an external interaction occurs is part of the characterization of the external interaction, at least in some cases, again making the specification of the required "exact effect" impractical.

16/5 {*AI12-0441-1*} An implementation that conforms to this Reference Manual shall support each capability required by the core language as specified. In addition, an implementation that conforms to this Reference Manual may conform to one or more Specialized Needs Annexes (or to none). Conformance to a Specialized Needs Annex means that each capability required by the Annex shall be provided as specified.

16.a **Discussion:** The last sentence defines what it means to say that an implementation conforms to a Specialized Needs Annex, namely, only by supporting all capabilities required by the Annex.

17/3 {*AI05-0229-1*} An implementation conforming to this Reference Manual may provide additional aspects, attributes, library units, and pragmas. However, it shall not provide any aspect, attribute, library unit, or pragma having the same name as an aspect, attribute, library unit, or pragma (respectively) specified in a Specialized Needs Annex unless the provided construct is either as specified in the Specialized Needs Annex or is more limited in capability than that required by the Annex. A program that attempts to use an unsupported capability of an Annex shall either be identified by the implementation before run time or shall raise an exception at run time.

17.a **Discussion:** The last sentence of the preceding paragraph defines what an implementation is allowed to do when it does not "conform" to a Specialized Needs Annex. In particular, the sentence forbids implementations from providing a construct with the same name as a corresponding construct in a Specialized Needs Annex but with a different syntax (e.g., an extended syntax) or quite different semantics. The phrase concerning "more limited in capability" is intended to give permission to provide a partial implementation, such as not implementing a subprogram in a package or having a restriction not permitted by an implementation that conforms to the Annex. For example, a partial implementation of the package Ada.Decimal might have Decimal.Max_Decimal_Digits as 15 (rather than the required 18). This allows a partial implementation to grow to a fully conforming implementation.

17.b A restricted implementation might be restricted by not providing some subprograms specified in one of the packages defined by an Annex. In this case, a program that tries to use the missing subprogram will usually fail to compile. Alternatively, the implementation might declare the subprogram as abstract, so it cannot be called. Alternatively, a subprogram body might be implemented just to raise Program_Error. The advantage of this approach is that a program to be run under a fully conforming Annex implementation can be checked syntactically and semantically under an implementation that only partially supports the Annex. Finally, an implementation might provide a package declaration without the corresponding body, so that programs can be compiled, but partitions cannot be built and executed.

17.c To ensure against wrong answers being delivered by a partial implementation, implementers are required to raise an exception when a program attempts to use an unsupported capability and this can be detected only at run time. For example, a partial implementation of Ada.Decimal might require the length of the Currency string to be 1, and hence, an exception would be raised if a subprogram were called in the package Edited_Output with a length greater than 1.

17.1/5 {*AI12-0179-1*} {*AI12-0265-1*} For an implementation that conforms to this Reference Manual, the implementation of a language-defined unit shall abide by all postconditions, type invariants, and default initial conditions specified for the unit by this document (see 11.4.2).

Documentation Requirements

18 Certain aspects of the semantics are defined to be either *implementation defined* or *unspecified*. In such cases, the set of possible effects is specified, and the implementation may choose any effect in the set.

Implementations shall document their behavior in implementation-defined situations, but documentation is not required for unspecified situations. The implementation-defined characteristics are summarized in M.2.

Discussion: We used to use the term “implementation dependent” instead of “unspecified”. However, that sounded too much like “implementation defined”. Furthermore, the term “unspecified” is used in the ANSI C and POSIX standards for this purpose, so that is another advantage. We also use “not specified” and “not specified by the language” as synonyms for “unspecified”. The documentation requirement is the only difference between implementation defined and unspecified. 18.a

Note that the “set of possible effects” can be “all imaginable effects”, as is the case with erroneous execution. 18.b

The implementation may choose to document implementation-defined behavior either by documenting what happens in general, or by providing some mechanism for the user to determine what happens in a particular case. 19

Discussion: For example, if the Reference Manual says that library unit elaboration order is implementation defined, the implementation might describe (in its user's manual) the algorithm it uses to determine the elaboration order. On the other hand, the implementation might provide a command that produces a description of the elaboration order for a partition upon request from the user. It is also acceptable to provide cross references to existing documentation (for example, a hardware manual), where appropriate. 19.a

Note that dependence of a program on implementation-defined or unspecified functionality is not defined to be an error; it might cause the program to be less portable, however. 19.b

Documentation Requirement: The behavior of implementations in implementation-defined situations shall be documented — see M.2 for a listing. 19.c/2

Implementation Advice

If an implementation detects the use of an unsupported Specialized Needs Annex feature at run time, it should raise Program_Error if feasible. 20

Implementation Advice: Program_Error should be raised when an unsupported Specialized Needs Annex feature is used at run time. 20.a.1/2

Reason: The reason we don't *require* Program_Error is that there are situations where other exceptions might make sense. For example, if the Real Time Systems Annex requires that the range of System.Priority include at least 30 values, an implementation could conform to the Reference Manual (but not to the Annex) if it supported only 12 values. Since the rules of the language require Constraint_Error to be raised for out-of-range values, we cannot require Program_Error to be raised instead. 20.a

If an implementation wishes to provide implementation-defined extensions to the functionality of a language-defined library unit, it should normally do so by adding children to the library unit. 21

Implementation Advice: Implementation-defined extensions to the functionality of a language-defined library unit should be provided by adding children to the library unit. 21.a.1/2

Implementation Note: If an implementation has support code (“run-time system code”) that is needed for the execution of user-defined code, it can put that support code in child packages of System. Otherwise, it has to use some trick to avoid polluting the user's namespace. It is important that such tricks not be available to user-defined code (not in the standard mode, at least) — that would defeat the purpose. 21.a

NOTE {*AI12-0440-1*} The above requirements imply that an implementation conforming to this Reference Manual can support some of the capabilities required by a Specialized Needs Annex without supporting all required capabilities. 22/5

Discussion: A conforming implementation can partially support a Specialized Needs Annex. Such an implementation does not conform to the Annex, but it does conform to the Reference Manual. 22.a

1.1.4 Method of Description and Syntax Notation

The form of an Ada program is described by means of a context-free syntax together with context-dependent requirements expressed by narrative rules. 1

The meaning of Ada programs is described by means of narrative rules defining both the effects of each construct and the composition rules for constructs. 2

3 The context-free syntax of the language is described using a simple variant of Backus-Naur Form. In particular:

4 • Lower case words in a sans-serif font, some containing embedded underlines, are used to denote syntactic categories, for example:

5 case_statement

6 • Boldface words are used to denote reserved words, for example:

7 **array**

8 • Square brackets enclose optional items. Thus the two following rules are equivalent.

9/2 {*AI95-00433-01*} simple_return_statement ::= **return** [expression];
simple_return_statement ::= **return**; | **return** expression;

10 • Curly brackets enclose a repeated item. The item may appear zero or more times; the repetitions occur from left to right as with an equivalent left-recursive rule. Thus the two following rules are equivalent.

11 term ::= factor {multiplying_operator factor}
term ::= factor | term multiplying_operator factor

12/5 • {*AI12-0212-1*} A vertical line separates alternative items, for example:

13/5 {*AI12-0212-1*} constraint ::= scalar_constraint | composite_constraint

13.1/5 • {*AI12-0212-1*} For symbols used in this notation (square brackets, curly brackets, and the vertical line), the symbols when surrounded by ' represent themselves, for example:

13.2/5 {*AI12-0212-1*} discrete_choice_list ::= discrete_choice {'|' discrete_choice}
named_container_aggregate ::= '[' container_element_association_list ']'

14 • If the name of any syntactic category starts with an italicized part, it is equivalent to the category name without the italicized part. The italicized part is intended to convey some semantic information. For example *subtype*_name and *task*_name are both equivalent to name alone.

14.a **Discussion:** The grammar given in this document is not LR(1). In fact, it is ambiguous; the ambiguities are resolved by the overload resolution rules (see 8.6).

14.b We often use “if” to mean “if and only if” in definitions. For example, if we define “photogenic” by saying, “A type is photogenic if it has the following properties...”, we mean that a type is photogenic if *and only if* it has those properties. It is usually clear from the context, and adding the “and only if” seems too cumbersome.

14.c When we say, for example, “a declarative_item of a declarative_part”, we are talking about a declarative_item immediately within that declarative_part. When we say “a declarative_item in, or within, a declarative_part”, we are talking about a declarative_item anywhere in the declarative_part, possibly deeply nested within other declarative_parts. (This notation doesn't work very well for names, since the name “of” something also has another meaning.)

14.d When we refer to the name of a language-defined entity (for example, Duration), we mean the language-defined entity even in programs where the declaration of the language-defined entity is hidden by another declaration. For example, when we say that the expected type for the expression of a delay_relative_statement is Duration, we mean the language-defined type Duration that is declared in Standard, not some type Duration the user might have declared.

14.1/3 {*AI95-00285-01*} {*AI05-0004-1*} {*AI05-0262-1*} The delimiters, compound delimiters, reserved words, and numeric_literals are exclusively made of the characters whose code point is between 16#20# and 16#7E#, inclusively. The special characters for which names are defined in this document (see 2.1) belong to the same range. [For example, the character E in the definition of exponent is the character whose name is “LATIN CAPITAL LETTER E”, not “GREEK CAPITAL LETTER EPSILON”.]

14.e/2 **Discussion:** This just means that programs can be written in plain ASCII characters; no characters outside of the 7-bit range are required.

{*AI95-00395-01*} {*AI05-0227-1*} {*AI05-0299-1*} {*AI12-0263-1*} {*AI12-0450-1*} When this document mentions the conversion of some character or sequence of characters to upper case, it means the character or sequence of characters obtained by using simple upper case mapping, as defined by documents referenced in Clause 2 of ISO/IEC 10646:2020. 14.2/5

Implementation Note: {*AI12-0263-1*} The "documents referenced" means Unicode, Chapter 4 (specifically, section 4.2 — Case). Machine-readable versions of Simple Uppercase Mapping and Simple Lowercase Mapping can be found in http://www.unicode.org/Public/UCD/latest/ucd/UnicodeData.txt. Data for older Unicode versions can be found on this site as well; start at http://www.unicode.org/Public/ and find the appropriate version number. Simple Uppercase Mapping is the 12th field in this file (the 13th element of each line, since Unicode counts from 0); the Simple Lowercase Mapping is the 13th field in this file. In both cases, if no character is present in the field, the character maps to itself. 14.f/5

A *syntactic category* is a nonterminal in the grammar defined in BNF under "Syntax". Names of syntactic categories are set in a different font, like_this. 15

A *construct* is a piece of text (explicit or implicit) that is an instance of a syntactic category defined under "Syntax". 16

Term entry: construct — piece of text (explicit or implicit) that is an instance of a syntactic category defined under Syntax 16.a/5

Ramification: For example, an expression is a construct. A declaration is a construct, whereas the thing declared by a declaration is an "entity". 16.b

Discussion: "Explicit" and "implicit" don't mean exactly what you might think they mean: The text of an instance of a generic is considered explicit, even though it does not appear explicitly (in the nontechnical sense) in the program text, and even though its meaning is not defined entirely in terms of that text. 16.c

A *constituent* of a construct is the construct itself, or any construct appearing within it. 17

Whenever the run-time semantics defines certain actions to happen in an *arbitrary order*, this means that the implementation shall arrange for these actions to occur in a way that is equivalent to some sequential order, following the rules that result from that sequential order. When evaluations are defined to happen in an arbitrary order, with conversion of the results to some subtypes, or with some runtime checks, the evaluations, conversions, and checks may be arbitrarily interspersed, so long as each expression is evaluated before converting or checking its value. [Note that the effect of a program can depend on the order chosen by the implementation. This can happen, for example, if two actual parameters of a given call have side effects.] 18

Discussion: Programs will be more portable if their external effect does not depend on the particular order chosen by an implementation. 18.a

Ramification: Additional reordering permissions are given in 11.6, "Exceptions and Optimization". 18.b

There is no requirement that the implementation always choose the same order in a given kind of situation. In fact, the implementation is allowed to choose a different order for two different executions of the same construct. However, we expect most implementations will behave in a relatively predictable manner in most situations. 18.c

Reason: The "sequential order" wording is intended to allow the programmer to rely on "benign" side effects. For example, if F is a function that returns a unique integer by incrementing some global and returning the result, a call such as P(F, F) is OK if the programmer cares only that the two results of F are unique; the two calls of F cannot be executed in parallel, unless the compiler can prove that parallel execution is equivalent to some sequential order. 18.d

NOTE 1 The syntax rules describing structured constructs are presented in a form that corresponds to the recommended paragraphing. For example, an if_statement is defined as: 19

```
if_statement ::=                                                    20
     if condition then
        sequence_of_statements
    {elsif condition then
        sequence_of_statements}
    [else
        sequence_of_statements]
     end if;
```

21 NOTE 2 The line breaks and indentation in the syntax rules indicate the recommended line breaks and indentation in the corresponding constructs. The preferred places for other line breaks are after semicolons.

Wording Changes from Ada 95

21.a/2 {*AI95-00285-01*} We now explicitly say that the lexical elements of the language (with a few exceptions) are made up of characters in the lower half of the Latin-1 character set. This is needed to avoid confusion given the new capability to use most ISO 10646 characters in identifiers and strings.

21.b/2 {*AI95-00395-01*} We now explicitly define what the Reference Manual means by upper case, as there are many possibilities for ISO 10646 characters.

21.c/2 {*AI95-00433-01*} The example for square brackets has been changed as there is no longer a return_statement syntax rule.

Wording Changes from Ada 2005

21.d/3 {*AI05-0227-1*} **Correction:** Upper case is defined by "simple upper case mapping", because "full case folding" is a mapping (mostly) to lower case.

Wording Changes from Ada 2012

21.e/5 {*AI12-0212-1*} Changed the syntax description to quote symbols used in the notation when they appear as themselves. We made this change to allow square brackets in Ada syntax, but the change also allowed us to get rid of an old hack which allowed vertical lines to stand for themselves in limited circumstances.

1.1.5 Classification of Errors

Language Design Principles

0.a/5 {*AI12-0204-1*} When the Reference Manual says that some construct *C1* has equivalent dynamic semantics to some other construct *C2*, then there should be a language rule that says that *C1* is illegal if *C2* is illegal.

0.b/5 **Reason:** {*AI12-0204-1*} We do not want to infer Legality Rules from Dynamic Semantics rules.

0.c/5 {*AI12-0064-2*} When the value of an expression is needed to enforce a Legality Rule or Post-Compilation Rule, then there should be a language rule requiring the expression to be static (see 4.9).

0.d/5 **Discussion:** {*AI12-0064-2*} This most likely occurs when the expression specifies the value of an aspect.

Implementation Requirements

1 The language definition classifies errors into several different categories:

2 • Errors that are required to be detected prior to run time by every Ada implementation;

3 These errors correspond to any violation of a rule given in this document, other than those listed below. In particular, violation of any rule that uses the terms shall, allowed, permitted, legal, or illegal belongs to this category. Any program that contains such an error is not a legal Ada program; on the other hand, the fact that a program is legal does not mean, *per se*, that the program is free from other forms of error.

4 The rules are further classified as either compile time rules, or post compilation rules, depending on whether a violation has to be detected at the time a compilation unit is submitted to the compiler, or may be postponed until the time a compilation unit is incorporated into a partition of a program.

4.a **Ramification:** See, for example, 10.1.3, “Subunits of Compilation Units”, for some errors that are detected only after compilation. Implementations are allowed, but not required, to detect post compilation rules at compile time when possible.

5 • Errors that are required to be detected at run time by the execution of an Ada program;

6 The corresponding error situations are associated with the names of the predefined exceptions. Every Ada compiler is required to generate code that raises the corresponding exception if such an error situation arises during program execution. [If such an error situation is certain to arise in

every execution of a construct, then an implementation is allowed (although not required) to report this fact at compilation time.]

- Bounded errors; 7

  {*AI12-0444-1*} The language rules define certain kinds of errors that are not expected to be detected either prior to or during run time, but if not detected, the range of possible effects shall be bounded. The errors of this category are called *bounded errors*. The possible effects of a given bounded error are specified for each such error, but in any case one possible effect of a bounded error is the raising of the exception Program_Error. 8/5

- Erroneous execution. 9

  {*AI12-0444-1*} In addition to bounded errors, the language rules define certain kinds of errors as leading to *erroneous execution*. Like bounded errors, the implementation is not expected to detect such errors either prior to or during run time. Unlike bounded errors, there is no language-specified bound on the possible effect of erroneous execution; the effect is in general not predictable. 10/5

  Ramification: Executions are erroneous, not programs or parts of programs. Once something erroneous happens, the execution of the entire program is erroneous from that point on, and potentially before given possible reorderings permitted by 11.6 and elsewhere. We cannot limit it to just one partition, since partitions are not required to live in separate address spaces. (But implementations are encouraged to limit it as much as possible.) 10.a

  Suppose a program contains a pair of things that will be executed "in an arbitrary order". It is possible that one order will result in something sensible, whereas the other order will result in erroneous execution. If the implementation happens to choose the first order, then the execution is not erroneous. This may seem odd, but it is not harmful. 10.b

  Saying that something is erroneous is semantically equivalent to saying that the behavior is unspecified. However, "erroneous" has a slightly more disapproving flavor. 10.c

Implementation Permissions

[An implementation may provide *nonstandard modes* of operation. Typically these modes would be selected by a pragma or by a command line switch when the compiler is invoked. When operating in a nonstandard mode, the implementation may reject compilation_units that do not conform to additional requirements associated with the mode, such as an excessive number of warnings or violation of coding style guidelines. Similarly, in a nonstandard mode, the implementation may apply special optimizations or alternative algorithms that are only meaningful for programs that satisfy certain criteria specified by the implementation. In any case, an implementation shall support a *standard* mode that conforms to the requirements of this Reference Manual; in particular, in the standard mode, all legal compilation_units shall be accepted.] 11

Discussion: These permissions are designed to authorize explicitly the support for alternative modes. Of course, nothing we say can prevent them anyway, but this (redundant) paragraph is designed to indicate that such alternative modes are in some sense "approved" and even encouraged where they serve the specialized needs of a given user community, so long as the standard mode, designed to foster maximum portability, is always available. 11.a

Implementation Advice

If an implementation detects a bounded error or erroneous execution, it should raise Program_Error. 12

Implementation Advice: If a bounded error or erroneous execution is detected, Program_Error should be raised. 12.a.1/2

Wording Changes from Ada 83

Some situations that are erroneous in Ada 83 are no longer errors at all. For example, depending on the parameter passing mechanism when unspecified is possibly nonportable, but not erroneous. 12.a

Other situations that are erroneous in Ada 83 are changed to be bounded errors. In particular, evaluating an uninitialized scalar variable is a bounded error. The possible results are to raise Program_Error (as always), or to produce a machine-representable value (which might not be in the subtype of the variable). Violating a Range_Check or Overflow_Check raises Constraint_Error, even if the value came from an uninitialized variable. This means that 12.b

optimizers can no longer “assume” that all variables are initialized within their subtype's range. Violating a check that is suppressed remains erroneous.

12.c The “incorrect order dependences” category of errors is removed. All such situations are simply considered potential nonportabilities. This category was removed due to the difficulty of defining what it means for two executions to have a “different effect”. For example, if a function with a side effect is called twice in a single expression, it is not in principle possible for the compiler to decide whether the correctness of the resulting program depends on the order of execution of the two function calls. A compile time warning might be appropriate, but raising of Program_Error at run time would not be.

1.2 Normative References

1/3 {*AI05-0299-1*} The following documents, in whole or in part, are normatively referenced in this document and are indispensable for its application. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies. For other documents mentioned in this document, see 1.2.1, “Bibliography”.

Paragraphs 1 through 7 and 9 through 10 have been moved to the Bibliography.

1.1/3 {*AI05-0127-2*} {*AI05-0299-1*} ISO 639-3:2007, *Codes for the representation of names of languages — Part 3: Alpha-3 code for comprehensive coverage of languages*

4.1/5 {*AI05-0127-2*} {*AI05-0299-1*} {*AI12-0450-1*} {*AI12-0454-1*} ISO 3166-1:2020, *Codes for the representation of names of countries and their subdivisions — Part 1: Country codes*

8/5 {*8652/0001*} {*AI95-00124-01*} {*AI95-00285-01*} {*AI05-0266-1*} {*AI12-0260-1*} {*AI12-0450-1*} ISO/IEC 10646:2020, *Information technology — Universal coded character set (UCS)*

Wording Changes from Ada 95

10.b/2 {*AI95-00285-01*} {*AI95-00376-01*} {*AI95-00415-01*} Updated references to the most recent versions of these standards. Added C++ and time standards. Added C character set technical report.

Wording Changes from Ada 2005

10.c/3 {*AI05-0127-2*} Added language and country code standards for locale support.

10.d/3 {*AI05-0266-1*} Updated references to the most recent versions of these standards.

Wording Changes from Ada 2012

10.e/5 {*AI12-0441-1*} {*AI12-0450-1*} Moved all informative references to the (new) 1.2.1, “Bibliography”. Revised remaining references to the most recent version.

1.2.1 Bibliography

1/5 {*AI12-0441-1*} The following documents are mentioned in this document as informative references.

2/5 {*AI12-0441-1*} ISO/IEC 646:1991, *Information technology — ISO 7-bit coded character set for information interchange*

3/5 {*AI12-0441-1*} ISO/IEC 1539-1:2018, *Information technology — Programming languages — Fortran — Part 1: Base language*

4/5 {*AI12-0441-1*} ISO/IEC 1989:2002, *Information technology — Programming languages — COBOL*

5/5 {*AI12-0441-1*} {*AI12-0450-1*} ISO/IEC 6429:1992, *Information technology — Control functions for coded character sets*

6/5 {*AI12-0441-1*} {*AI12-0450-1*} {*AI12-0454-1*} ISO 8601-1:2019, *Date and time — Representations for information interchange — Part 1: Basic rules*

{*AI12-0441-1*} ISO/IEC 8859-1:1998, *Information technology — 8-bit single-byte coded graphic character sets — Part 1: Latin alphabet No. 1* 7/5

{*AI12-0441-1*} {*AI12-0450-1*} ISO/IEC 9899:2018, *Information technology — Programming languages — C* 8/5

{*AI12-0441-1*} {*AI12-0450-1*} {*AI12-0454-1*} ISO/IEC 14882:2020, *Programming languages — C++* 9/5

{*AI12-0437-1*} ISO/IEC 60559:2020, *Information technology — Microprocessor Systems — Floating-Point arithmetic* 10/5

Discussion: {*AI12-0437-1*} POSIX, *Portable Operating System Interface (POSIX) — Part 1: System Application Program Interface (API) [C Language]*, The Institute of Electrical and Electronics Engineers, 1990 10.a/5

Wording Changes from Ada 2012

{*AI12-0441-1*} {*AI12-0450-1*} Moved the majority of references here, updating the dates and names for some. 10.b/5

1.3 Terms and Definitions

{*AI95-00415-01*} {*AI12-0443-1*} Terms are defined throughout this document, indicated by *italic* type. Terms explicitly defined in this document are not to be presumed to refer implicitly to similar terms defined elsewhere. Mathematical terms not defined in this document are to be interpreted according to the *CRC Concise Encyclopedia of Mathematics, Second Edition*. Other terms not defined in this document are to be interpreted according to the *Webster's Third New International Dictionary of the English Language*. Informal descriptions of some terms are also given below. 1/5

Discussion: The index contains an entry for every defined term. 1.a

{*AI95-00415-01*} The contents of the *CRC Concise Encyclopedia of Mathematics, Second Edition* can be accessed on http://www.mathworld.com. The ISBN number of the book is ISBN 1584883472. 1.b/2

{*AI12-0443-1*} The definitions found in this subclause are short, descriptive definitions. These are most useful for comprehension when an exact meaning is not needed. More formal, detailed definitions are found in the body of the Reference Manual; these are needed to determine the exact application of rules to constructs and entities. 1.c/5

{*AI12-0443-1*} To see the difference, consider “aliased view”. The descriptive definition is “a view of an object that can be designated by an access value”. The formal definition is a lengthy paragraph in 3.10 that defines 10 separate constructs as aliased views. In order to reason about a language rule that requires an aliased view in a particular context, one needs to look at the list of constructs in order to determine if the object in hand is actually an aliased view. The descriptive definition is no help in this case. 1.d/5

Term entry: {*AI12-0443-1*} Each term defined in this subclause is marked like this in the body of the AARM near its detailed formal definition. 1.e/5

Discussion: Here are some AARM-only definitions: The Ada Rapporteur Group (ARG) interprets the Ada Reference Manual. An Ada Issue (AI) is a numbered ruling from the ARG. Ada Issues created for Ada 83 are denoted as "AI83", while Ada Issues created for Ada 95 are denoted as "AI95" in this document. Similarly, Ada Issues created for Ada 2005 are denoted as "AI05", Ada Issues created for Ada 2012 are denoted as "AI12", and Ada Issues created for Ada 2022 are denoted as "AI22". The Ada Commentary Integration Document (ACID) is an edition of the Ada 83 RM in which clearly marked insertions and deletions indicate the effect of integrating the approved AIs. The Uniformity Rapporteur Group (URG) issued recommendations intended to increase uniformity across Ada implementations. The functions of the URG have been assumed by the ARG. A Uniformity Issue (UI) was a numbered recommendation from the URG. A Defect Report and Response is an official query to WG9 about an error in the standard. Defect Reports are processed by the ARG, and are referenced here by their ISO numbers: 8652/nnnn. Most changes to the Ada 95 standard include reference(s) to the Defect Report(s) that prompted the change. The *Ada Conformity Assessment Test Suite (ACATS)* is a set of tests intended to check the conformity of Ada implementations to this standard. This set of tests was previously known as the Ada Compiler Validation Capability (ACVC). 1.f/5

1.3.1 Types, Objects, and their Properties

1/5 **abstract type**
tagged type intended for use as an ancestor of other types, but which is not allowed to have objects of its own

2/5 **access type**
type that has values that designate aliased objects

3/5 Note: Access types correspond to “pointer types” or “reference types” in some other languages.

4/5 **accessibility level**
representation of the lifetime of an entity in terms of the level of dynamic nesting within which the entity is known to exist

5/5 **aliased view**
view of an object that can be designated by an access value

6/5 Note: Objects allocated by allocators are aliased. Objects can also be explicitly declared as aliased with the reserved word aliased. The Access attribute can be used to create an access value designating an aliased object.

7/5 **ancestor of a type**
type itself or, in the case of a type derived from other types, its parent type or one of its progenitor types or one of their ancestors

8/5 Note: Ancestor and descendant are inverse relationships.

9/5 **array type**
composite type whose components are all of the same type

10/5 **aspect**
specifiable property of an entity

11/5 Note: An aspect can be specified by an aspect_specification on the declaration of the entity. Some aspects can be queried via attributes.

12/5 **attribute**
characteristic or property of an entity that can be queried, and in some cases specified

13/5 **category of types**
set of types with one or more common properties, such as primitive operations

14/5 Note: A category of types that is closed under derivation is also known as a class.

15/5 **character type**
enumeration type whose values include characters

16/5 **class of types**
set of types that is closed under derivation, which means that if a given type is in the class, then all types derived from that type are also in the class

17/5 Note: The set of types of a class share common properties, such as their primitive operations.

18/5 **composite type**
type with components, such as an array or record

19/5 **controlled type**
type that supports user-defined assignment and finalization

20/5 Note: Objects are always finalized before being destroyed.

21/5 **default initial condition**
property that holds for every default-initialized object of a given type

derived type 22/5
type defined in terms of a parent type and zero or more progenitor types given in a derived type definition

Note 1: A derived type inherits properties such as components and primitive operations from its parent and progenitors. 23/5

Note 2: A type together with the types derived from it (directly or indirectly) form a derivation class. 24/5

descendant of a type 25/5
type itself or a type derived (directly or indirectly) from it

Note: Descendant and ancestor are inverse relationships. 26/5

discrete type 27/5
type that is either an integer type or an enumeration type

discriminant 28/5
parameter for a composite type, which can control, for example, the bounds of a component that is an array

Note: A discriminant for a task type can be used to pass data to a task of the type upon its creation. 29/5

elementary type 30/5
type that does not have components

enumeration type 31/5
type defined by an enumeration of its values, which can be denoted by identifiers or character literals

full type 32/5
type that defines a full view

full view 33/5
view of a type that reveals all of its properties

Note: There can be other views of the type that reveal fewer properties. 34/5

incomplete type 35/5
type that defines an incomplete view

Note: Incomplete types can be used for defining recursive data structures. 36/5

incomplete view 37/5
view of a type that reveals minimal properties

Note: The remaining properties are defined by the full view given elsewhere. 38/5

indexable container type 39/5
type that has user-defined behavior for indexing, via the Constant_Indexing or Variable_Indexing aspects

integer type 40/5
type that represents signed or modular integers

Note: A signed integer type has a base range that includes both positive and negative numbers, and has operations that can raise an exception when the result is outside the base range. A modular type has a base range whose lower bound is zero, and has operations with “wraparound” semantics. Modular types subsume what are called “unsigned types” in some other languages. 41/5

interface type 42/5
abstract tagged type that has no components or concrete operations except possibly null procedures

Note: Interface types are used for composing other interfaces and tagged types and thereby provide multiple inheritance. Only an interface type can be used as a progenitor of another type. 43/5

invariant 44/5
assertion that is expected to be True for all objects of a given private type when viewed from outside the defining package

45/5 **iterable container type**
type that has user-defined behavior for iteration, via the Default_Iterator and Iterator_Element aspects

46/5 **limited type**
type for which copying (such as in an assignment_statement) is not allowed

47/5 Note: All types are either limited types or nonlimited types.

48/5 **needed component**
component of a record type or record extension that is required to have its value specified within a given aggregate

49/5 **nominal subtype**
subtype specified when a view of an object is defined

50/5 **nonlimited type**
type for which copying is allowed

51/5 **object**
entity that contains a value, and is either a constant or a variable

52/5 Note: An object is created by an object_declaration or by an allocator. A formal parameter is (a view of) an object. A subcomponent of an object is an object.

53/5 **operational aspect**
aspect that indicates a logical property of an entity, such as the precondition of a subprogram, or the procedure used to write a given type of object to a stream

54/5 **parent of a derived type**
first ancestor type given in the definition of the derived type

55/5 Note: The parent can be almost any kind of type, including an interface type.

56/5 **partial view**
view of a type that reveals only some of its properties

57/5 Note: The remaining properties are defined by the full view given elsewhere.

58/5 **primitive operations of a type**
operations (such as subprograms) declared together with the type declarations

59/5 Note: Primitive operations are inherited by other types in the same derivation class of types.

60/5 **private extension**
type that extends another type, with the additional properties hidden from its clients

61/5 **private type**
type that defines a partial view

62/5 Note: Private types can be used for defining abstractions that hide unnecessary details from their clients.

63/5 **progenitor**
type given in the interface list, if any, of an interface, task, protected, or derived type definition

64/5 Note: A progenitor is always an interface type.

65/5 **protected type**
composite type whose components are accessible only through one of its protected operations, which synchronize concurrent access by multiple tasks

66/5 **real type**
type that has values that are approximations of the real numbers

67/5 Note: Floating point and fixed point types are real types.

68/5 **record extension**
type that extends another type optionally with additional components

record type 69/5
composite type consisting of zero or more named components, possibly of different types

reference type 70/5
type that has user-defined behavior for ".all", defined by the Implicit_Dereference aspect

representation aspect 71/5
aspect that indicates how an entity is mapped onto the underlying hardware, for example the size or alignment of an object

scalar type 72/5
either a discrete type or a real type

stable property 73/5
characteristic associated with objects of a given type that is preserved by many of the primitive operations of the type

storage pool object 74/5
object associated with one or more access types from which the storage for objects created by allocators of the access type(s) is obtained

Note: Some storage pools can be partitioned into subpools in order to support finer-grained storage management. 75/5

stream 76/5
sequence of elements that can be used, along with the stream-oriented attributes, to support marshalling and unmarshalling of values of most types

subtype 77/5
type together with optional constraints, null exclusions, and predicates, which constrain the values of the type to the subset that satisfies the implied conditions

synchronized 78/5
can be safely operated on by multiple tasks concurrently

Note: Synchronized is used to qualify entities, as in a synchronized interface. 79/5

tagged type 80/5
type whose objects each have a run-time type tag, which indicates the specific type for which the object was originally created

Note: Tagged types can be extended with additional components. 81/5

task type 82/5
composite type used to represent active entities which execute concurrently and that can communicate via queued task entries

Note: The top-level task of a partition is called the environment task. 83/5

type 84/5
defining characteristic of each object and expression of the language, with an associated set of values, and a set of primitive operations that implement the fundamental aspects of its semantics

Note: Types are grouped into categories. Most language-defined categories of types are also classes of types. 85/5

view of an entity 86/5
representation of an entity that reveals some or all of the properties of the entity

Note: A single entity can have multiple views. 87/5

1.3.2 Subprograms and their Properties

function 1/5
form of subprogram that returns a result and can be called as part of an expression

2/5 **overriding operation**
operation that replaces an inherited primitive operation

3/5 Note: Operations can be marked explicitly as overriding or not overriding.

4/5 **postcondition**
assertion that is expected to be True when a given subprogram returns normally

5/5 **precondition**
assertion that is expected to be True when a given subprogram is called

6/5 **procedure**
form of subprogram that does not return a result and can only be invoked by a statement

7/5 **subprogram**
unit of a program that can be brought into execution in various contexts, with the invocation being a subprogram call that can parameterize the effect of the subprogram through the passing of operands

8/5 Note: There are two forms of subprograms: functions, which return values, and procedures, which do not.

1.3.3 Other Syntactic Constructs

1/5 **aggregate**
construct used to define a value of a composite type by specifying the values of the components of the type

2/5 **compilation unit**
program unit that is separately compiled

3/5 Note: A compilation_unit contains either the declaration, the body, or a renaming of a program unit.

4/5 **construct**
piece of text (explicit or implicit) that is an instance of a syntactic category defined under Syntax

5/5 **container**
structured object that represents a collection of elements all of the same (potentially class-wide) type, such as a vector or a tree

6/5 Note: Several predefined container types are provided by the children of package Ada.Containers (see A.18.1).

7/5 **container aggregate**
construct used to define a value of a type that represents a collection of elements, by explicitly specifying the elements in the collection

8/5 **core language**
clause or annex in which are defined language constructs or capabilities that are provided by all conforming implementations

9/5 Note: A construct is said to be part of the core language if it is defined in a core language clause or annex.

10/5 **declaration**
language construct that associates a name with (a view of) an entity

11/5 Note: A declaration can appear explicitly in the program text (an explicit declaration), or can be supposed to occur at a given place in the text as a consequence of the semantics of another construct (an implicit declaration).

12/5 **generic instance**
nongeneric unit created by the instantiation of a generic unit

13/5 **generic unit**
template for a (nongeneric) program unit

14/5 Note 1: The template can be parameterized by objects, types, subprograms, and packages.

15/5 Note 2: Generic units can be used to perform the role that macros sometimes play in other languages.

iterator 16/5
construct that is used to loop over the elements of an array or container

Note: Iterators can be user defined, and can perform arbitrary computations to access elements from a container. 17/5

iterator filter 18/5
construct that is used to restrict the elements produced by an iteration to those for which a boolean condition evaluates to True

library unit 19/5
separately compiled program unit, which is a package, a subprogram, or a generic unit

Note: Library units can have other (logically nested) library units as children, and can have other program units physically nested within them. A root library unit, together with its children and grandchildren and so on, form a subsystem. 20/5

master construct 21/5
one of certain executable constructs for which there can be objects or tasks whose lifetime ends when the construct completes

Note: Execution of a master construct is a master, with which objects and tasks are associated for the purposes of waiting and finalization. 22/5

needed compilation unit 23/5
compilation unit that is necessary to produce an executable partition, because some entity declared or defined within the unit is used elsewhere in the partition

package 24/5
program unit that defines the interface to a group of logically related entities, along with their implementation

Note: Typically, a package contains the declaration of a type (often a private type or private extension) along with the declarations of primitive subprograms of the type, which can be called from outside the package, while their inner workings remain hidden from outside users. 25/5

parallel construct 26/5
executable construct that defines multiple activities of a single task that can proceed in parallel, via the execution of multiple logical threads of control

partition 27/5
part of a program, which consists of a set of interdependent library units

Note: Each partition can run in a separate address space, possibly on a separate computer. A program can contain just one partition, or it can be distributed across multiple partitions, which can execute concurrently. 28/5

pragma 29/5
compiler directive to provide control over and above that provided by the other syntactic constructs of the language

Note: There are language-defined pragmas that give instructions for optimization, listing control, etc. An implementation can support additional (implementation-defined) pragmas. 30/5

program 31/5
set of partitions, each of which can execute in a separate address space, possibly on a separate computer

program unit 32/5
language construct that is a package, a task unit, a protected unit, a protected entry, a generic unit, or an explicitly declared subprogram other than an enumeration literal

Note: Certain kinds of program units can be separately compiled. Alternatively, they can appear physically nested within other program units. 33/5

reduction expression 34/5
expression that defines how to map or transform a collection of values into a new set of values, and then summarize the values by applying an operation to reduce the set to a single value

35/5 **renaming**
declaration that does not define a new entity, but instead defines a new view of an existing entity

36/5 **specialized needs annex**
annex in which are defined language constructs or capabilities that are not necessarily provided by all conforming implementations

37/5 **subunit**
body of a program unit that can be compiled separately from its enclosing program unit

1.3.4 Runtime Actions

1/5 **assertion**
boolean expression that is expected to be True at run time at certain specified places

2/5 Note: Certain pragmas and aspects define various kinds of assertions.

3/5 **elaboration**
process by which a declaration achieves its run-time effect

4/5 Note: Elaboration is one of the forms of execution.

5/5 **evaluation**
process by which an expression achieves its run-time effect

6/5 Note: Evaluation is one of the forms of execution.

7/5 **execution**
process by which a construct achieves its run-time effect

8/5 Note: Execution of a declaration is also called elaboration. Execution of an expression is also called evaluation.

9/5 **logical thread of control**
activity within the execution of a program that can proceed in parallel with other activities of the same task, or of separate tasks

10/5 **master**
execution of a master construct

11/5 Note: Each object and task is associated with a master. When a master is left, associated tasks are awaited and associated objects are finalized.

1.3.5 Exceptional Situations

1/5 **check**
test made during execution to determine whether a language rule has been violated

2/5 **exception**
kind of exceptional situation

3/5 **exception occurrence**
run-time occurrence of an exceptional situation

4/5 **handle an exception**
perform some actions in response to the arising of an exception

5/5 **raise an exception**
abandon normal program execution so as to draw attention to the fact that the corresponding situation has arisen

6/5 **suppress a check**
assert that the check cannot fail, and request that the compiler optimize by disabling the check

Note: The compiler is not required to honor this request. Suppressing checks that can fail can cause a program to behave in arbitrary ways. 7/5

2 Lexical Elements

{*AI05-0299-1*} [The text of a program consists of the texts of one or more compilations. The text of a compilation is a sequence of lexical elements, each composed of characters; the rules of composition are given in this clause. Pragmas, which provide certain information for the compiler, are also described in this clause.] 1/3

2.1 Character Set

{*AI95-00285-01*} {*AI95-00395-01*} {*AI05-0266-1*} {*AI12-0263-1*} {*AI12-0450-1*} {*AI12-0454-1*} The character repertoire for the text of an Ada program consists of the entire coding space described by the ISO/IEC 10646:2020 Universal Coded Character Set. This coding space is organized in *planes*, each plane comprising 65536 characters. 1/5

This paragraph was deleted.{*AI95-00285-01*} 1.a/2

This paragraph was deleted.{*AI95-00285-01*} 1.b/2

Discussion: {*AI95-00285-01*} {*AI05-0266-1*} {*AI12-0263-1*} {*AI12-0450-1*} It is our intent to follow the terminology of ISO/IEC 10646:2020 where appropriate, and to remain compatible with the character classifications defined in A.3, “Character Handling”. 1.c/5

Syntax

Paragraphs 2 and 3 were deleted.

{*AI95-00285-01*} {*AI95-00395-01*} {*AI05-0266-1*} {*AI12-0263-1*} {*AI12-0450-1*} A character is defined by this Reference Manual for each cell in the coding space described by ISO/IEC 10646:2020, regardless of whether or not ISO/IEC 10646:2020 allocates a character to that cell. 3.1/5

Static Semantics

{*AI95-00285-01*} {*AI95-00395-01*} {*AI05-0079-1*} {*AI05-0262-1*} {*AI05-0266-1*} {*AI12-0263-1*} {*AI12-0444-1*} {*AI12-0450-1*} The coded representation for characters is implementation defined [(it can be a representation that is not defined within ISO/IEC 10646:2020)]. A character whose relative code point in its plane is 16#FFFE# or 16#FFFF# is not allowed anywhere in the text of a program. The only characters allowed outside of comments are those in categories other_format, format_effector, and graphic_character. 4/5

Implementation defined: The coded representation for the text of an Ada program. 4.a

Ramification: {*AI95-00285-01*} Note that this rule doesn't really have much force, since the implementation can represent characters in the source in any way it sees fit. For example, an implementation could simply define that what seems to be an other_private_use character is actually a representation of the space character. 4.b/2

{*AI95-00285-01*} {*AI05-0266-1*} {*AI05-0299-1*} {*AI12-0004-1*} {*AI12-0263-1*} {*AI12-0450-1*} The semantics of an Ada program whose text is not in Normalization Form C (as defined by Clause 22 of ISO/IEC 10646:2020) is implementation defined. 4.1/5

Implementation defined: The semantics of an Ada program whose text is not in Normalization Form C. 4.c/5

Ramification: {*AI12-0004-1*} In particular, an implementation can reject such program source. It is easy during lexical analysis to reject source that contains any code point not present in Normalization Form C. Portable programs should always be encoded in Normalization Form C. 4.d/5

Reason: {*AI12-0004-1*} Normalization Form C ensures that all source is in a unique format; it eliminates ambiguities and security issues potentially caused by source using unusual sequences of characters. Note that WC3 (the Internet standards group) recommends that all Internet content be in Normalization Form C. We don't require this as there is a potentially significant cost to checking this (just rejecting unallowed code points is not enough), and some implementations may need to be interoperable with tools that produce unnormalized text. 4.e/5

5/5 {*AI95-00285-01*} {*AI05-0266-1*} {*AI05-0299-1*} {*AI12-0263-1*} {*AI12-0450-1*} The description of the language definition in this document uses the character properties General Category, Simple Uppercase Mapping, Uppercase Mapping, and Special Case Condition of the documents referenced by Clause 2 of ISO/IEC 10646:2020. The actual set of graphic symbols used by an implementation for the visual representation of the text of an Ada program is not specified.

5.a/5 **Discussion:** {*AI12-0263-1*} The "documents referenced" means Unicode, Chapter 4. See the Discussion after the the character categorization definition for a source for machine-readable definitions of these properties.

6/3 {*AI95-00285-01*} {*AI05-0266-1*} Characters are categorized as follows:

6.a/5 **Discussion:** {*AI05-0005-1*} {*AI05-0262-1*} {*AI05-0266-1*} {*AI12-0263-1*} {*AI12-0450-1*} Our character classification considers that the cells not allocated in ISO/IEC 10646:2020 are graphic characters, except for those whose relative code point in their plane is 16#FFFE# or 16#FFFF#. This seems to provide the best compatibility with future versions of ISO/IEC 10646, as future characters can already be used in Ada character and string literals.

7/2 *This paragraph was deleted.*{*AI95-00285-01*}

8/2 {*AI95-00285-01*} letter_uppercase
Any character whose General Category is defined to be "Letter, Uppercase".

9/2 {*AI95-00285-01*} letter_lowercase
Any character whose General Category is defined to be "Letter, Lowercase".

9.a/1 *This paragraph was deleted.*{*8652/0001*} {*AI95-00124-01*}

9.1/2 {*AI95-00285-01*} letter_titlecase
Any character whose General Category is defined to be "Letter, Titlecase".

9.2/2 {*AI95-00285-01*} letter_modifier
Any character whose General Category is defined to be "Letter, Modifier".

9.3/2 {*AI95-00285-01*} letter_other
Any character whose General Category is defined to be "Letter, Other".

9.4/2 {*AI95-00285-01*} mark_non_spacing
Any character whose General Category is defined to be "Mark, Non-Spacing".

9.5/2 {*AI95-00285-01*} mark_spacing_combining
Any character whose General Category is defined to be "Mark, Spacing Combining".

10/2 {*AI95-00285-01*} number_decimal
Any character whose General Category is defined to be "Number, Decimal".

10.1/2 {*AI95-00285-01*} number_letter
Any character whose General Category is defined to be "Number, Letter".

10.2/2 {*AI95-00285-01*} punctuation_connector
Any character whose General Category is defined to be "Punctuation, Connector".

10.3/2 {*AI95-00285-01*} other_format
Any character whose General Category is defined to be "Other, Format".

11/2 {*AI95-00285-01*} separator_space
Any character whose General Category is defined to be "Separator, Space".

12/2 {*AI95-00285-01*} separator_line
Any character whose General Category is defined to be "Separator, Line".

12.1/2 {*AI95-00285-01*} separator_paragraph
Any character whose General Category is defined to be "Separator, Paragraph".

13/3 {*AI95-00285-01*} {*AI05-0262-1*} format_effector
The characters whose code points are 16#09# (CHARACTER TABULATION), 16#0A# (LINE FEED), 16#0B# (LINE TABULATION), 16#0C# (FORM FEED), 16#0D#

(CARRIAGE RETURN), 16#85# (NEXT LINE), and the characters in categories separator_line and separator_paragraph.

Discussion: ISO/IEC 10646:2003 does not define the names of control characters, but rather refers to the names defined by ISO/IEC 6429:1992. These are the names that we use here. 13.a/2

{*AI95-00285-01*} other_control 13.1/2

Any character whose General Category is defined to be “Other, Control”, and which is not defined to be a format_effector.

{*AI95-00285-01*} other_private_use 13.2/2

Any character whose General Category is defined to be “Other, Private Use”.

{*AI95-00285-01*} other_surrogate 13.3/2

Any character whose General Category is defined to be “Other, Surrogate”.

{*AI95-00285-01*} {*AI95-00395-01*} {*AI05-0262-1*} graphic_character 14/3

Any character that is not in the categories other_control, other_private_use, other_surrogate, format_effector, and whose relative code point in its plane is neither 16#FFFE# nor 16#FFFF#.

This paragraph was deleted. 14.a/2

Discussion: {*AI95-00285-01*} We considered basing the definition of lexical elements on Annex A of ISO/IEC TR 10176 (4th edition), which lists the characters which should be supported in identifiers for all programming languages, but we finally decided against this option. Note that it is not our intent to diverge from ISO/IEC TR 10176, except to the extent that ISO/IEC TR 10176 itself diverges from ISO/IEC 10646:2003 (which is the case at the time of this writing [January 2005]). 14.b/2

More precisely, we intend to align strictly with ISO/IEC 10646:2003. It must be noted that ISO/IEC TR 10176 is a Technical Report while ISO/IEC 10646:2003 is a Standard. If one has to make a choice, one should conform with the Standard rather than with the Technical Report. And, it turns out that one *must* make a choice because there are important differences between the two: 14.c/2

- ISO/IEC TR 10176 is still based on ISO/IEC 10646:2000 while ISO/IEC 10646:2003 has already been published for a year. We cannot afford to delay the adoption of our amendment until ISO/IEC TR 10176 has been revised. 14.d/2
- There are considerable differences between the two editions of ISO/IEC 10646, notably in supporting characters beyond the BMP (this might be significant for some languages, e.g. Korean). 14.e/2
- ISO/IEC TR 10176 does not define case conversion tables, which are essential for a case-insensitive language like Ada. To get case conversion tables, we would have to reference either ISO/IEC 10646:2003 or Unicode, or we would have to invent our own. 14.f/2

For the purpose of defining the lexical elements of the language, we need character properties like categorization, as well as case conversion tables. These are mentioned in ISO/IEC 10646:2003 as useful for implementations, with a reference to Unicode. Machine-readable tables are available on the web at URLs: 14.g/2

```
http://www.unicode.org/Public/UCD/latest/ucd/UnicodeData.txt
http://www.unicode.org/Public/UCD/latest/ucd/CaseFolding.txt
```
14.h/2

with an explanatory document found at URL: 14.i/2

```
http://www.unicode.org/reports/tr44/tr44-20.html
```
14.j/2

{*AI12-0263-1*} The actual text of the standard only makes specific references to the corresponding clauses of ISO/IEC 10646, not to Unicode. 14.k/5

15/5 {*AI95-00285-01*} {*AI05-0266-1*} {*AI12-0263-1*} {*AI12-0450-1*} The following names are used when referring to certain characters (the first name is that given in ISO/IEC 10646:2020):

15.a/5 **Discussion:** {*AI95-00285-01*} {*AI05-0266-1*} {*AI12-0125-3*} {*AI12-0212-1*} {*AI12-0263-1*} {*AI12-0450-1*} This table serves to show the correspondence between ISO/IEC 10646:2020 names and the graphic symbols (glyphs) used in this document. These are the characters that play a special role in the syntax of Ada.

graphic symbol	name	graphic symbol	name
"	quotation mark	:	colon
#	number sign	;	semicolon
&	ampersand	<	less-than sign
'	apostrophe, tick	=	equals sign
(	left parenthesis	>	greater-than sign
)	right parenthesis	_	low line, underline
*	asterisk, multiply	|	vertical line
+	plus sign	/	solidus, divide
,	comma	!	exclamation point
–	hyphen-minus, minus	%	percent sign
.	full stop, dot, point	[	left square bracket
@	commercial at, at sign	]	right square bracket

Implementation Requirements

16/3 {*AI05-0286-1*} An Ada implementation shall accept Ada source code in UTF-8 encoding, with or without a BOM (see A.4.11), where every character is represented by its code point. The character pair CARRIAGE RETURN/LINE FEED (code points 16#0D# 16#0A#) signifies a single end of line (see 2.2); every other occurrence of a format_effector other than the character whose code point position is 16#09# (CHARACTER TABULATION) also signifies a single end of line.

16.a/3 **Reason:** {*AI05-0079-1*} {*AI05-0286-1*} This is simply requiring that an Ada implementation be able to directly process the ACATS, which is provided in the described format. Note that files that only contain characters with code points in the first 128 (which is the majority of the ACATS) are represented in the same way in both UTF-8 and in "plain" string format. The ACATS includes a BOM in files that have any characters with code points greater than 127. Note that the BOM contains characters not legal in Ada source code, so an implementation can use that to automatically distinguish between files formatted as plain Latin-1 strings and UTF-8 with BOM.

16.b/3 We allow line endings to be both represented as the pair CR LF (as in Windows and the ACATS), and as single format_effector characters (usually LF, as in Linux), in order that files created by standard tools on most operating systems will meet the standard format. We specify how many line endings each represent so that compilers use the same line numbering for standard source files.

16.c/3 This requirement increases portability by having a format that is accepted by all Ada compilers. Note that implementations can support other source representations, including structured representations like a parse tree.

Implementation Permissions

17/3 {*AI95-00285-01*} {*AI05-0266-1*} The categories defined above, as well as case mapping and folding, may be based on an implementation-defined version of ISO/IEC 10646 (2003 edition or later).

17.b/3 **Ramification:** The exact categories, case mapping, and case folding chosen affects identifiers, the result of '[[Wide_]Wide_]Image, and packages Wide_Characters.Handling and Wide_Wide_Characters.Handling.

17.c/3 **Discussion:** This permission allows implementations to upgrade to using a newer character set standard whenever that makes sense, rather than having to wait for the next Ada Standard. But the character set standard used cannot be older than ISO/IEC 10646:2003 (which is essentially similar to Unicode 4.0).

18/2 NOTE {*AI95-00285-01*} The characters in categories other_control, other_private_use, and other_surrogate are only allowed in comments.

19.a/3 *This paragraph was deleted.*{*AI05-0286-1*}

Extensions to Ada 83

Ada 95 allows 8-bit and 16-bit characters, as well as implementation-specified character sets. 19.b

Wording Changes from Ada 83

{*AI95-00285-01*} {*AI05-0299-1*} The syntax rules in this subclause are modified to remove the emphasis on basic characters vs. others. (In this day and age, there is no need to point out that you can write programs without using (for example) lower case letters.) In particular, character (representing all characters usable outside comments) is added, and basic_graphic_character, other_special_character, and basic_character are removed. Special_character is expanded to include Ada 83's other_special_character, as well as new 8-bit characters not present in Ada 83. Ada 2005 removes special_character altogether; we want to stick to ISO/IEC 10646:2003 character classifications. Note that the term “basic letter” is used in A.3, “Character Handling” to refer to letters without diacritical marks. 19.c/3

{*AI95-00285-01*} Character names now come from ISO/IEC 10646:2003. 19.d/2

This paragraph was deleted.{*AI95-00285-01*} 19.e/2

Extensions to Ada 95

{*AI95-00285-01*} {*AI95-00395-01*} Program text can use most characters defined by ISO-10646:2003. This subclause has been rewritten to use the categories defined in that Standard. This should ease programming in languages other than English. 19.f/2

Inconsistencies With Ada 2005

{*AI05-0299-1*} {*AI05-0266-1*} An implementation is allowed (but not required) to use a newer character set standard to determine the categories, case mapping, and case folding. Doing so will change the results of attributes '[[Wide_]Wide_]Image and the packages [Wide_]Wide_Characters.Handling in the case of a few rarely used characters. (This also could make some identifiers illegal, for characters that are no longer classified as letters.) This is unlikely to be a problem in practice. Moreover, truly portable Ada 2012 programs should avoid using in these contexts any characters that would have different classifications in any character set standards issued since 10646:2003 (since the compiler can use any such standard as the basis for its classifications). 19.g/3

Wording Changes from Ada 2005

{*AI05-0079-1*} **Correction:** Clarified that only characters in the categories defined here are allowed in the source of an Ada program. This was clear in Ada 95, but Amendment 1 dropped the wording instead of correcting it. 19.h/3

{*AI05-0286-1*} A standard source representation is defined that all compilers are expected to process. Since this is the same format as the ACATS, it seems unlikely that there are any implementations that don't meet this requirement. Moreover, other representations are still permitted, and the "impossible or impractical" loophole (see 1.1.3) can be invoked for any implementations that cannot directly process the ACATS. 19.i/3

Wording Changes from Ada 2012

{*AI12-0004-1*} **Correction:** The interpretation of Ada source that is in Normalization Form C but not in Normalization Form KC is no longer implementation-defined. This change could potentially change the meaning of a program for a compiler that normalized all program source to Normalization Form KC before processing it. We don't document this as an inconsistency as such handling was previously implementation defined (so any such code was already defined to be not portable), and we're not aware of any compiler that normalized source code (so we don't expect to see this problem in the real world). 19.j/5

2.2 Lexical Elements, Separators, and Delimiters

Static Semantics

The text of a program consists of the texts of one or more compilations. The text of each compilation is a sequence of separate *lexical elements*. Each lexical element is formed from a sequence of characters, and is either a delimiter, an identifier, a reserved word, a numeric_literal, a character_literal, a string_literal, or a comment. The meaning of a program depends only on the particular sequences of lexical elements that form its compilations, excluding comments. 1

{*AI95-00285-01*} {*AI05-0262-1*} The text of a compilation is divided into *lines*. In general, the representation for an end of line is implementation defined. However, a sequence of one or more 2/3

format_effectors other than the character whose code point is 16#09# (CHARACTER TABULATION) signifies at least one end of line.

2.a **Implementation defined:** The representation for an end of line.

3/2 {*AI95-00285-01*} [In some cases an explicit *separator* is required to separate adjacent lexical elements.] A separator is any of a separator_space, a format_effector, or the end of a line, as follows:

4/2 - {*AI95-00285-01*} A separator_space is a separator except within a comment, a string_literal, or a character_literal.

5/3 - {*AI95-00285-01*} {*AI05-0262-1*} The character whose code point is 16#09# (CHARACTER TABULATION) is a separator except within a comment.

6 - The end of a line is always a separator.

7 One or more separators are allowed between any two adjacent lexical elements, before the first of each compilation, or after the last. At least one separator is required between an identifier, a reserved word, or a numeric_literal and an adjacent identifier, reserved word, or numeric_literal.

7.1/3 {*AI05-0079-1*} One or more other_format characters are allowed anywhere that a separator is[; any such characters have no effect on the meaning of an Ada program].

8/2 {*AI95-00285-01*} A *delimiter* is either one of the following characters:

9/5 {*AI12-0125-3*} {*AI12-0212-1*} & ' ( ) * + , – . / : ; < = > @ [ ] |

10 or one of the following *compound delimiters* each composed of two adjacent special characters

11 => .. ** := /= >= <= << >> <>

12 Each of the special characters listed for single character delimiters is a single delimiter except if this character is used as a character of a compound delimiter, or as a character of a comment, string_literal, character_literal, or numeric_literal.

13 The following names are used when referring to compound delimiters:

delimiter	name
=>	arrow
..	double dot
**	double star, exponentiate
:=	assignment (pronounced: “becomes”)
/=	inequality (pronounced: “not equal”)
>=	greater than or equal
<=	less than or equal
<<	left label bracket
>>	right label bracket
<>	box

Implementation Requirements

14 An implementation shall support lines of at least 200 characters in length, not counting any characters used to signify the end of a line. An implementation shall support lexical elements of at least 200

characters in length. The maximum supported line length and lexical element length are implementation defined.

Implementation defined: Maximum supported line length and lexical element length. 14.a

Discussion: From URG recommendation. 14.b

Wording Changes from Ada 95

{*AI95-00285-01*} {*AI05-0299-1*} The wording was updated to use the new character categories defined in the preceding subclause. 14.c/3

Extensions to Ada 2005

{*AI05-0079-1*} **Correction:** Clarified that other_format characters are allowed anywhere that separators are allowed. This was intended in Ada 2005, but didn't actually make it into the wording. 14.d/3

Wording Changes from Ada 2012

{*AI12-0125-3*} {*AI12-0212-1*} Added square brackets and the target_name symbol (see 5.2.1) to the list of delimiters. 14.e/5

2.3 Identifiers

Identifiers are used as names. 1

Syntax

{*AI95-00285-01*} {*AI95-00395-01*} identifier ::= 2/2
  identifier_start {identifier_start | identifier_extend}

{*AI95-00285-01*} {*AI95-00395-01*} identifier_start ::= 3/2
    letter_uppercase
  | letter_lowercase
  | letter_titlecase
  | letter_modifier
  | letter_other
  | number_letter

{*AI95-00285-01*} {*AI95-00395-01*} {*AI05-0091-1*} identifier_extend ::= 3.1/3
    mark_non_spacing
  | mark_spacing_combining
  | number_decimal
  | punctuation_connector

{*AI95-00395-01*} {*AI05-0091-1*} An identifier shall not contain two consecutive characters in category punctuation_connector, or end with a character in that category. 4/3

Reason: This rule was stated in the syntax in Ada 95, but that has gotten too complex in Ada 2005. 4.a/3

Legality Rules

{*AI12-0004-1*} {*AI12-0263-1*} {*AI12-0450-1*} An identifier shall only contain characters that may be present in Normalization Form KC as defined by Clause 22 of ISO/IEC 10646:2020. 4.1/5

Implementation Note: {*AI12-0004-1*} {*AI12-0005-1*} An implementation can usually detect this during lexical processing. The code points not allowed are those for which Unicode property NFKC_QC (Normalization Form KC Quick_Check) has the value No. We say "may be present" so that characters for which the value is Maybe (really, one of the possible values is Maybe) are allowed (these are mainly combining marks). The necessary tables can be found in http://www.unicode.org/Public/UCD/latest/ucd/DerivedNormalizationProps.txt. Versions for older Unicode versions can be found on this site as well; start at http://www.unicode.org/Public/ and find the appropriate version number. 4.b/5

Static Semantics

5/5 {*AI95-00285-01*} {*AI05-0091-1*} {*AI05-0227-1*} {*AI05-0266-1*} {*AI05-0299-1*} {*AI12-0263-1*} {*AI12-0450-1*} Two identifiers are considered the same if they consist of the same sequence of characters after applying locale-independent simple case folding, as defined by documents referenced in Clause 2 of ISO/IEC 10646:2020.

5.a/3 **Discussion:** {*AI05-0227-1*} Simple case folding is a mapping to lower case, so this is matching the defining (lower case) version of a reserved word. We could have mentioned case folding of the reserved words, but as that is an identity function, it would have no effect.

5.a.1/5 {*AI05-0227-1*} {*AI12-0263-1*} The "documents referenced" means Unicode, Chapter 4 (specifically, section 4.2 — Case). Note that simple case folding is supposed to be compatible between Unicode versions, so the Unicode version used doesn't matter. A machine-readable version of the needed mapping can be found at: http://www.unicode.org/Public/UCD/latest/ucd/CaseFolding.txt.

5.3/3 {*AI95-00395-01*} {*AI05-0091-1*} {*AI05-0227-1*} After applying simple case folding, an identifier shall not be identical to a reserved word.

5.b/3 **Implementation Note:** We match the reserved words after applying case folding so that the rules for identifiers and reserved words are the same. Since a compiler usually will lexically process identifiers and reserved words the same way (often with the same code), this will prevent a lot of headaches.

5.c/3 **Ramification:** {*AI05-0227-1*} The rules for reserved words differ in one way: they define case conversion on letters rather than sequences. This means that it is possible that there exist some unusual sequences that are neither identifiers nor reserved words. We are not aware of any such sequences so long as we use simple case folding (as opposed to full case folding), but we have defined the rules in case any are introduced in future character set standards. This originally was a problem when converting to upper case: "ıf" and "acceß" have upper case conversions of "IF" and "ACCESS" respectively. We would not want these to be treated as reserved words. But neither of these cases exist when using simple case folding.

Implementation Permissions

6 In a nonstandard mode, an implementation may support other upper/lower case equivalence rules for identifiers[, to accommodate local conventions].

6.a/3 **Discussion:** {*AI95-00285-01*} {*AI05-0227-1*} For instance, in most languages, the simple case folded equivalent of LATIN CAPITAL LETTER I (an upper case letter without a dot above) is LATIN SMALL LETTER I (a lower case letter with a dot above). In Turkish, though, LATIN CAPITAL LETTER I and LATIN CAPITAL LETTER I WITH DOT ABOVE are two distinct letters, so the case folded equivalent of LATIN CAPITAL LETTER I is LATIN SMALL LETTER DOTLESS I, and the case folded equivalent of LATIN CAPITAL LETTER I WITH DOT ABOVE is LATIN SMALL LETTER I. Take for instance the following identifier (which is the name of a city on the Tigris river in Eastern Anatolia):

6.b/3 `DİYARBAKIR -- The first i is dotted, the second isn't.`

6.c/3 A Turkish reader would expect that the above identifier is equivalent to:

6.d/3 `diyarbakır`

6.d.1/3 However, locale-independent simple case folding (and thus Ada) maps this to:

6.d.2/3 `dİyarbakir`

6.e/3 which is different from any of the following identifiers:

6.f/2
```
diyarbakir
diyarbakır
dıyarbakir
dıyarbakır
```

6.f.1/3 including the "correct" matching identifier for Turkish. Upper case conversion (used in '[Wide_]Wide_Image) introduces additional problems.

6.g/3 An implementation targeting the Turkish market is allowed (in fact, expected) to provide a nonstandard mode where case folding is appropriate for Turkish.

6.j/2 Lithuanian and Azeri are two other languages that present similar idiosyncrasies.

6.1/2 NOTE {*AI95-00285-01*} Identifiers differing only in the use of corresponding upper and lower case letters are considered the same.

Examples

Examples of identifiers: 7

```
{AI95-00433-01} Count      X    Get_Symbol   Ethelyn   Marion                    8/2
Snobol_4   X1   Page_Count   Store_Next_Item
Πλάτων       -- Plato
Чайковский   -- Tchaikovsky
θ  φ         -- Angles
```

Wording Changes from Ada 83

We no longer include reserved words as identifiers. This is not a language change. In Ada 83, identifier included reserved words. However, this complicated several other rules (for example, regarding implementation-defined attributes and pragmas, etc.). We now explicitly allow certain reserved words for attribute designators, to make up for the loss. 8.a

Ramification: Because syntax rules are relevant to overload resolution, it means that if it looks like a reserved word, it is not an identifier. As a side effect, implementations cannot use reserved words as implementation-defined attributes or pragma names. 8.b

Extensions to Ada 95

{*AI95-00285-01*} An identifier can use any letter defined by ISO-10646:2003, along with several other categories. This should ease programming in languages other than English. 8.c/2

Incompatibilities With Ada 2005

{*AI05-0091-1*} **Correction:** other_format characters were removed from identifiers as the Unicode recommendations have changed. This change can only affect programs written for the original Ada 2005, so there should be few such programs. 8.d/3

{*AI05-0227-1*} **Correction:** We now specify simple case folding rather than full case folding. That potentially could change identifier equivalence, although it is more likely that identifiers that are considered the same in original Ada 2005 will now be considered different. This change was made because the original Ada 2005 definition was incompatible (and even inconsistent in unusual cases) with the Ada 95 identifier equivalence rules. As such, the Ada 2005 rules were rarely fully implemented, and in any case, only Ada 2005 identifiers containing wide characters could be affected. 8.e/3

Incompatibilities With Ada 2012

{*AI12-0004-1*} **Correction:** An identifier that contains any characters not permitted in Normalization Form KC is now illegal. Ada 2012 allowed such identifiers, but their interpretation was implementation-defined (so the use of such identifiers was not portable). Identifiers that may be interpreted differently by different compilers are a safety and security hazard, so we no longer allow them. 8.f/5

2.4 Numeric Literals

There are two kinds of numeric_literals, *real literals* and *integer literals*. A real literal is a numeric_literal that includes a point; an integer literal is a numeric_literal without a point. 1

Syntax

numeric_literal ::= decimal_literal | based_literal 2

NOTE The type of an integer literal is *universal_integer*. The type of a real literal is *universal_real*. 3

2.4.1 Decimal Literals

A decimal_literal is a numeric_literal in the conventional decimal notation (that is, the base is ten). 1

Syntax

decimal_literal ::= numeral [.numeral] [exponent] 2

numeral ::= digit {[underline] digit} 3

4 exponent ::= E [+] numeral | E – numeral

4.1/2 {*AI95-00285-01*} digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

5 An exponent for an integer literal shall not have a minus sign.

5.a **Ramification:** Although this rule is in this subclause, it applies also to the next subclause.

Static Semantics

6 An underline character in a numeric_literal does not affect its meaning. The letter E of an exponent can be written either in lower case or in upper case, with the same meaning.

6.a **Ramification:** Although these rules are in this subclause, they apply also to the next subclause.

7 An exponent indicates the power of ten by which the value of the decimal_literal without the exponent is to be multiplied to obtain the value of the decimal_literal with the exponent.

Examples

8 *Examples of decimal literals:*

9
```
12          0       1E6     123_456     -- integer literals
12.0        0.0     0.456   3.14159_26  -- real literals
```

Wording Changes from Ada 83

9.a We have changed the syntactic category name integer to be numeral. We got this idea from ACID. It avoids the confusion between this and integers. (Other places don't offer similar confusions. For example, a string_literal is different from a string.)

2.4.2 Based Literals

1 [A based_literal is a numeric_literal expressed in a form that specifies the base explicitly.]

Syntax

2 based_literal ::=
  base # based_numeral [.based_numeral] # [exponent]

3 base ::= numeral

4 based_numeral ::=
  extended_digit {[underline] extended_digit}

5 extended_digit ::= digit | A | B | C | D | E | F

Legality Rules

6 The *base* (the numeric value of the decimal numeral preceding the first #) shall be at least two and at most sixteen. The extended_digits A through F represent the digits ten through fifteen, respectively. The value of each extended_digit of a based_literal shall be less than the base.

Static Semantics

7 The conventional meaning of based notation is assumed. An exponent indicates the power of the base by which the value of the based_literal without the exponent is to be multiplied to obtain the value of the based_literal with the exponent. The base and the exponent, if any, are in decimal notation.

8 The extended_digits A through F can be written either in lower case or in upper case, with the same meaning.

Examples

Examples of based literals: 9

```
2#1111_1111#  16#FF#      016#0ff#   -- integer literals of value 255
16#E#E1       2#1110_0000#           -- integer literals of value 224
16#F.FF#E+2   2#1.1111_1111_1110#E11 -- real literals of value 4095.0
```
10

Wording Changes from Ada 83

The rule about which letters are allowed is now encoded in BNF, as suggested by Mike Woodger. This is clearly more readable. 10.a

2.5 Character Literals

[A character_literal is formed by enclosing a graphic character between two apostrophe characters.] 1

Syntax

character_literal ::= 'graphic_character' 2

NOTE A character_literal is an enumeration literal of a character type. See 3.5.2. 3

Examples

Examples of character literals: 4

```
{AI95-00433-01} 'A'     '*'     '''     ' '
'L'     'Л'     'Λ'     -- Various els.
'∞'     'א'             -- Big numbers - infinity and aleph.
```
5/2

Wording Changes from Ada 83

{*AI05-0299-1*} The definitions of the values of literals are in Clauses 3 and 4, rather than here, since it requires knowledge of types. 5.a/3

2.6 String Literals

[A string_literal is formed by a sequence of graphic characters (possibly none) enclosed between two quotation marks used as string brackets. They are used to represent operator_symbols (see 6.1), values of a string type (see 4.2), and array subaggregates (see 4.3.3).] 1

Syntax

string_literal ::= "{string_element}" 2

string_element ::= "" | *non_quotation_mark*_graphic_character 3

A string_element is either a pair of quotation marks (""), or a single graphic_character other than a quotation mark. 4

Static Semantics

The *sequence of characters* of a string_literal is formed from the sequence of string_elements between the bracketing quotation marks, in the given order, with a string_element that is "" becoming a single quotation mark in the sequence of characters, and any other string_element being reproduced in the sequence. 5

A *null string literal* is a string_literal with no string_elements between the quotation marks. 6

NOTE 1 An end of line cannot appear in a string_literal. 7

NOTE 2 {*AI95-00285-01*} No transformation is performed on the sequence of characters of a string_literal. 7.1/2

Examples

8 *Examples of string literals:*

9/2
```
{AI95-00433-01} "Message of the day:"

""                      -- a null string literal
" "   "A"   """"        -- three string literals of length 1

"Characters such as $, %, and } are allowed in string literals"
"Archimedes said ""Εύρηκα"""
"Volume of cylinder (πr²h) = "
```

Wording Changes from Ada 83

9.a The wording has been changed to be strictly lexical. No mention is made of string or character values, since string_literals are also used to represent operator_symbols, which don't have a defined value.

9.b The syntax is described differently.

Wording Changes from Ada 95

9.c/2 {*AI95-00285-01*} We explicitly say that the characters of a string_literal should be used as is. In particular, no normalization or folding should be performed on a string_literal.

2.7 Comments

1 A comment starts with two adjacent hyphens and extends up to the end of the line.

Syntax

2 comment ::= --{*non_end_of_line*_character}

3 A comment may appear on any line of a program.

Static Semantics

4 The presence or absence of comments has no influence on whether a program is legal or illegal. Furthermore, comments do not influence the meaning of a program; their sole purpose is the enlightenment of the human reader.

Examples

5 *Examples of comments:*

6/5
```
{AI12-0440-1} -- the last sentence above echoes the Algol 68 report

end;  -- processing of Line is complete

-- a long comment can be split onto
-- two or more consecutive lines

---------------- the first two hyphens start the comment
```

2.8 Pragmas

1 A pragma is a compiler directive. There are language-defined pragmas that give instructions for optimization, listing control, etc. An implementation may support additional (implementation-defined) pragmas.

1.a/5 **Term entry: pragma** — compiler directive to provide control over and above that provided by the other syntactic constructs of the language
Note: There are language-defined pragmas that give instructions for optimization, listing control, etc. An implementation can support additional (implementation-defined) pragmas.

Language Design Principles

{*AI05-0100-1*} {*AI05-0163-1*} In general, if all pragmas are treated as unrecognized pragmas, the program should remain both syntactically and semantically legal. There are a few exceptions to this general principle (for example, pragma Import can eliminate the need for a completion), but the principle remains, and is strictly true at the syntactic level. Certainly any implementation-defined pragmas should obey this principle both syntactically and semantically, so that if the pragmas are not recognized by some other implementation, the program will remain legal. 1.b/3

Syntax

pragma ::= 2
  pragma identifier [(pragma_argument_association {, pragma_argument_association})];

{*AI05-0290-1*} pragma_argument_association ::= 3/3
    [*pragma_argument_*identifier =>] name
  | [*pragma_argument_*identifier =>] expression
  | *pragma_argument_*aspect_mark => name
  | *pragma_argument_*aspect_mark => expression

{*AI05-0290-1*} In a pragma, any pragma_argument_associations without a *pragma_argument_*identifier or *pragma_argument_*aspect_mark shall precede any associations with a *pragma_argument_*identifier or *pragma_argument_*aspect_mark. 4/3

Pragmas are only allowed at the following places in a program: 5

- {*AI12-0236-1*} After a semicolon delimiter, but not within a formal_part, discriminant_part, or declare_expression. 6/5
- {*AI05-0100-1*} {*AI05-0163-1*} At any place where the syntax rules allow a construct defined by a syntactic category whose name ends with "declaration", "item", "statement", "clause", or "alternative", or one of the syntactic categories variant or exception_handler; but not in place of such a construct if the construct is required, or is part of a list that is required to have at least one such construct. 7/3
- {*AI05-0163-1*} In place of a statement in a sequence_of_statements. 7.1/3
- {*AI05-0100-1*} At any place where a compilation_unit is allowed. 7.2/3

Additional syntax rules and placement restrictions exist for specific pragmas. 8

Discussion: The above rule is written in text, rather than in BNF; the syntactic category pragma is not used in any BNF syntax rule. 8.a

Ramification: A pragma is allowed where a generic_formal_parameter_declaration is allowed. 8.b

The *name* of a pragma is the identifier following the reserved word **pragma**. The name or expression of a pragma_argument_association is a *pragma argument*. 9

To be honest: {*AI95-00284-02*} For compatibility with Ada 83, the name of a pragma may also be "**interface**", which is not an identifier (because it is a reserved word). See J.12. 9.a/2

{*AI05-0272-1*} An *identifier specific to a pragma* is an identifier or reserved word that is used in a pragma argument with special meaning for that pragma. 10/3

To be honest: Whenever the syntax rules for a given pragma allow "identifier" as an argument of the pragma, that identifier is an identifier specific to that pragma. 10.a

{*AI05-0272-1*} In a few cases, a reserved word is allowed as "an identifier specific to a pragma". Even in these cases, the syntax still is written as identifier (the reserved word(s) are not shown). For example, the restriction No_Use_Of_Attribute (see 13.12.1) allows the reserved words which can be attribute designators, but the syntax for a restriction does not include these reserved words. 10.b/3

Static Semantics

11 If an implementation does not recognize the name of a pragma, then it has no effect on the semantics of the program. Inside such a pragma, the only rules that apply are the Syntax Rules.

11.a **To be honest:** This rule takes precedence over any other rules that imply otherwise.

11.b **Ramification:** Note well: this rule applies only to pragmas whose name is not recognized. If anything else is wrong with a pragma (at compile time), the pragma is illegal. This is true whether the pragma is language defined or implementation defined.

11.c For example, an expression in an unrecognized pragma does not cause freezing, even though the rules in 13.14, "Freezing Rules" say it does; the above rule overrules those other rules. On the other hand, an expression in a recognized pragma causes freezing, even if this makes something illegal.

11.d For another example, an expression that would be ambiguous is not illegal if it is inside an unrecognized pragma.

11.e Note, however, that implementations have to recognize **pragma** Inline(Foo) and freeze things accordingly, even if they choose to never do inlining.

11.f Obviously, the contradiction needs to be resolved one way or the other. The reasons for resolving it this way are: The implementation is simple — the compiler can just ignore the pragma altogether. The interpretation of constructs appearing inside implementation-defined pragmas is implementation defined. For example: "**pragma** Mumble(X);". If the current implementation has never heard of Mumble, then it doesn't know whether X is a name, an expression, or an identifier specific to the pragma Mumble.

11.g **To be honest:** The syntax of individual pragmas overrides the general syntax for pragma.

11.h **Ramification:** Thus, an identifier specific to a pragma is not a name, syntactically; if it were, the visibility rules would be invoked, which is not what we want.

11.i/3 {*AI05-0229-1*} This also implies that named associations do not allow one to give the arguments in an arbitrary order — the order given in the syntax rule for each individual pragma must be obeyed. However, it is generally possible to leave out earlier arguments when later ones are given; for example, this is allowed by the syntax rule for pragma Import (see J.15.5, "Interfacing Pragmas"). As for subprogram calls, positional notation precedes named notation.

11.j Note that Ada 83 had no pragmas for which the order of named associations mattered, since there was never more than one argument that allowed named associations.

11.k **To be honest:** The interpretation of the arguments of implementation-defined pragmas is implementation defined. However, the syntax rules have to be obeyed.

Dynamic Semantics

12 Any pragma that appears at the place of an executable construct is executed. Unless otherwise specified for a particular pragma, this execution consists of the evaluation of each evaluable pragma argument in an arbitrary order.

12.a **Ramification:** For a pragma that appears at the place of an elaborable construct, execution is elaboration.

12.b An identifier specific to a pragma is neither a name nor an expression — such identifiers are not evaluated (unless an implementation defines them to be evaluated in the case of an implementation-defined pragma).

12.c The "unless otherwise specified" part allows us (and implementations) to make exceptions, so a pragma can contain an expression that is not evaluated. Note that pragmas in type_definitions may contain expressions that depend on discriminants.

12.d When we wish to define a pragma with some run-time effect, we usually make sure that it appears in an executable context; otherwise, special rules are needed to define the run-time effect and when it happens.

Implementation Requirements

13 The implementation shall give a warning message for an unrecognized pragma name.

13.a **Ramification:** An implementation is also allowed to have modes in which a warning message is suppressed, or in which the presence of an unrecognized pragma is a compile-time error.

Implementation Permissions

14 An implementation may provide implementation-defined pragmas; the name of an implementation-defined pragma shall differ from those of the language-defined pragmas.

Implementation defined: Implementation-defined pragmas. 14.a

Ramification: The semantics of implementation-defined pragmas, and any associated rules (such as restrictions on their placement or arguments), are, of course, implementation defined. Implementation-defined pragmas may have run-time effects. 14.b

An implementation may ignore an unrecognized pragma even if it violates some of the Syntax Rules, if detecting the syntax error is too complex. 15

Reason: Many compilers use extra post-parsing checks to enforce the syntax rules, since the Ada syntax rules are not LR(k) (for any k). (The grammar is ambiguous, in fact.) This paragraph allows them to ignore an unrecognized pragma, without having to perform such post-parsing checks. 15.a

Implementation Advice

{*AI05-0163-1*} Normally, implementation-defined pragmas should have no semantic effect for error-free programs; that is, if the implementation-defined pragmas in a working program are replaced with unrecognized pragmas, the program should still be legal, and should still have the same semantics. 16/3

Implementation Advice: Implementation-defined pragmas should have no semantic effect for error-free programs. 16.a.1/2

Ramification: Note that "semantics" is not the same as "effect;" as explained in 1.1.3, the semantics defines a set of possible effects. 16.a

Note that adding a pragma to a program might cause an error (either at compile time or at run time). On the other hand, if the language-specified semantics for a feature are in part implementation defined, it makes sense to support pragmas that control the feature, and that have real semantics; thus, this paragraph is merely a recommendation. 16.b

Normally, an implementation should not define pragmas that can make an illegal program legal, except as follows: 17

- {*AI05-0229-1*} A pragma used to complete a declaration; 18/3

  Discussion: {*AI05-0229-1*} There are no language-defined pragmas which can be completions; pragma Import was defined this way in Ada 95 and Ada 2005, but in Ada 2012 pragma Import just sets aspect Import which disallows having any completion. 18.a/3

- A pragma used to configure the environment by adding, removing, or replacing library_items. 19

  Implementation Advice: Implementation-defined pragmas should not make an illegal program legal, unless they complete a declaration or configure the library_items in an environment. 19.a.1/2

  Ramification: For example, it is OK to support Interface, System_Name, Storage_Unit, and Memory_Size pragmas for upward compatibility reasons, even though all of these pragmas can make an illegal program legal. (The latter three can affect legality in a rather subtle way: They affect the value of named numbers in System, and can therefore affect the legality in cases where static expressions are required.) 19.a

  On the other hand, adding implementation-defined pragmas to a legal program can make it illegal. For example, a common kind of implementation-defined pragma is one that promises some property that allows more efficient code to be generated. If the promise is a lie, it is best if the user gets an error message. 19.b

  Incompatibilities With Ada 83

  In Ada 83, "bad" pragmas are ignored. In Ada 95, they are illegal, except in the case where the name of the pragma itself is not recognized by the implementation. 19.c

  Extensions to Ada 83

  Implementation-defined pragmas may affect the legality of a program. 19.d

  Wording Changes from Ada 83

  Implementation-defined pragmas may affect the run-time semantics of the program. This was always true in Ada 83 (since it was not explicitly forbidden by RM83), but it was not clear, because there was no definition of "executing" or "elaborating" a pragma. 19.e

  Extensions to Ada 2005

  {*AI05-0163-1*} **Correction:** Allow pragmas in place of a statement, even if there are no other statements in a sequence_of_statements. 19.f/3

19.g/3 {*AI05-0272-1*} Identifiers specific to a pragma can be reserved words.

19.h/3 {*AI05-0290-1*} Pragma arguments can be identified with aspect_marks; this allows identifier'Class in this context. As usual, this is only allowed if specifically allowed by a particular pragma.

Wording Changes from Ada 2005

19.i/3 {*AI05-0100-1*} **Correction:** Clarified where pragmas are (and are not) allowed.

Wording Changes from Ada 2012

19.j/5 {*AI12-0236-1*} Added wording to ensure that pragmas are not allowed in declare_expressions. We don't allow this as the definition of most pragmas assume that they're given between entities. We also don't want to answer questions about conformance of pragmas, especially unrecognized pragmas and pragmas whose state can change between the specification and body.

Syntax

20 The forms of List, Page, and Optimize pragmas are as follows:

21 **pragma** List(identifier);

22 **pragma** Page;

23 **pragma** Optimize(identifier);

24 [Other pragmas are defined throughout this Reference Manual, and are summarized in Annex L.]

24.a **Ramification:** The language-defined pragmas are supported by every implementation, although “supporting” some of them (for example, Inline) requires nothing more than checking the arguments, since they act only as advice to the implementation.

Static Semantics

25 A pragma List takes one of the identifiers On or Off as the single argument. This pragma is allowed anywhere a pragma is allowed. It specifies that listing of the compilation is to be continued or suspended until a List pragma with the opposite argument is given within the same compilation. The pragma itself is always listed if the compiler is producing a listing.

26 A pragma Page is allowed anywhere a pragma is allowed. It specifies that the program text which follows the pragma should start on a new page (if the compiler is currently producing a listing).

27 A pragma Optimize takes one of the identifiers Time, Space, or Off as the single argument. This pragma is allowed anywhere a pragma is allowed, and it applies until the end of the immediately enclosing declarative region, or for a pragma at the place of a compilation_unit, to the end of the compilation. It gives advice to the implementation as to whether time or space is the primary optimization criterion, or that optional optimizations should be turned off. [It is implementation defined how this advice is followed.]

27.a **Implementation defined:** Effect of pragma Optimize.

27.b **Discussion:** For example, a compiler might use Time vs. Space to control whether generic instantiations are implemented with a macro-expansion model, versus a shared-generic-body model.

27.c We don't define what constitutes an “optimization” — in fact, it cannot be formally defined in the context of Ada. One compiler might call something an optional optimization, whereas another compiler might consider that same thing to be a normal part of code generation. Thus, the programmer cannot rely on this pragma having any particular portable effect on the generated code. Some compilers might even ignore the pragma altogether.

Examples

Examples of pragmas: 28

```
{AI95-00433-01} {AI05-0229-1} {AI12-0417-1} pragma List(Off); -- turn off listing generation
pragma Optimize(Off); -- turn off optional optimizations
pragma Assertion_Policy(Check); -- check assertions
pragma Assert(Exists(File_Name),
              Message => "Nonexistent file"); -- assert file exists
```
29/5

Extensions to Ada 83

The Optimize pragma now allows the identifier Off to request that normal optimization be turned off. 29.a

An Optimize pragma may appear anywhere pragmas are allowed. 29.b

Wording Changes from Ada 83

We now describe the pragmas Page, List, and Optimize here, to act as examples, and to remove the normative material from Annex L, “Language-Defined Pragmas”, so it can be entirely an informative annex. 29.c

Wording Changes from Ada 95

{*AI95-00433-01*} Updated the example of named pragma parameters, because the second parameter of pragma Suppress is obsolescent. 29.d/2

Wording Changes from Ada 2005

{*AI05-0229-1*} Updated the example of pragmas, because both pragmas Inline and Import are obsolescent. 29.e/3

2.9 Reserved Words

Syntax

This paragraph was deleted. 1/1

{*AI95-00284-02*} {*AI95-00395-01*} {*AI05-0091-1*} The following are the *reserved words*. Within a program, some or all of the letters of a reserved word may be in upper case. 2/3

Discussion: Reserved words have special meaning in the syntax. In addition, certain reserved words are used as attribute names. 2.a

The syntactic category identifier no longer allows reserved words. We have added the few reserved words that are legal explicitly to the syntax for attribute_reference. Allowing identifier to include reserved words has been a source of confusion for some users, and differs from the way they are treated in the C and Pascal language definitions. 2.b

{*AI12-0005-1*} Reserved words are sometimes called *keywords* in informal material. 2.c/5

abort	**else**	**new**	**return**
abs	**elsif**	**not**	**reverse**
abstract	**end**	**null**	**select**
accept	**entry**	**of**	**separate**
access	**exception**	**or**	**some**
aliased	**exit**	**others**	**subtype**
all	**for**	**out**	**synchronized**
and	**function**	**overriding**	**tagged**
array	**generic**	**package**	**task**
at	**goto**	**parallel**	**terminate**
begin	**if**	**pragma**	**then**
body	**in**	**private**	**type**
case	**interface**	**procedure**	**until**
constant	**is**	**protected**	**use**
declare	**limited**	**raise**	**when**
delay	**loop**	**range**	**while**
delta	**mod**	**record**	**with**
digits		**rem**	**xor**
do		**renames**	
		requeue	

3/5 NOTE {*AI12-0440-1*} The reserved words appear in **lower case boldface** in this document, except when used in the designator of an attribute (see 4.1.4). Lower case boldface is also used for a reserved word in a string_literal used as an operator_symbol. This is merely a convention — programs can be written in whatever typeface is desired and available.

Incompatibilities With Ada 83

3.a The following words are not reserved in Ada 83, but are reserved in Ada 95: **abstract**, **aliased**, **protected**, **requeue**, **tagged**, **until**.

Wording Changes from Ada 83

3.b/3 {*AI05-0299-1*} The subclause entitled “Allowed Replacements of Characters” has been moved to Annex J, “Obsolescent Features”.

Incompatibilities With Ada 95

3.c/2 {*AI95-00284-02*} The following words are not reserved in Ada 95, but are reserved in Ada 2005: **interface**, **overriding**, **synchronized**. A special allowance is made for **pragma** Interface (see J.12). Uses of these words as identifiers will need to be changed, but we do not expect them to be common.

Wording Changes from Ada 95

3.d/2 {*AI95-00395-01*} The definition of upper case equivalence has been modified to allow identifiers using all of the characters of ISO 10646. This change has no effect on the character sequences that are reserved words, but does make some unusual sequences of characters illegal.

Incompatibilities With Ada 2005

3.e/3 {*AI05-0091-1*} **Correction:** Removed other_format characters from reserved words in order to be compatible with the latest Unicode recommendations. This change can only affect programs written for original Ada 2005, and there is little reason to put other_format characters into reserved words in the first place, so there should be very few such programs.

3.f/3 {*AI05-0176-1*} The following word is not reserved in Ada 2005, but is reserved in Ada 2012: **some**. Uses of this word as an identifier will need to be changed, but we do not expect them to be common.

Incompatibilities With Ada 2012

{*AI12-0119-1*} The following word is not reserved in Ada 2012, but is reserved in Ada 2022: **parallel**. Uses of this word as an identifier will need to be changed, but we do not expect them to be common. 3.g/5

3 Declarations and Types

{*AI05-0299-1*} This clause describes the types in the language and the rules for declaring constants, variables, and named numbers. 1/3

3.1 Declarations

{*AI12-0373-1*} The language defines several kinds of named *entities* that are declared by declarations. The entity's *name* is defined by the declaration, usually by a defining_identifier, but sometimes by a defining_character_literal or defining_operator_symbol. There are also entities that are not directly declared; some of these are elements of other entities, or are allocated dynamically. Such entities can be denoted using indexed_component, selected_component, or dereference names (see 4.1). 1/5

Discussion: {*AI12-0373-1*} Some entities are always anonymous. For instance, a type is never named (the name represents the first subtype). We don't mention those here as this paragraph is about *named* entities. 1.a/5

There are several forms of declaration. A basic_declaration is a form of declaration defined as follows. 2

Syntax

{*AI95-00348-01*} {*AI05-0177-1*} basic_declaration ::= 3/3

```
    type_declaration          | subtype_declaration
  | object_declaration        | number_declaration
  | subprogram_declaration    | abstract_subprogram_declaration
  | null_procedure_declaration | expression_function_declaration
  | package_declaration       | renaming_declaration
  | exception_declaration     | generic_declaration
  | generic_instantiation
```

defining_identifier ::= identifier 4

Static Semantics

A *declaration* is a language construct that associates a name with (a view of) an entity. A declaration may appear explicitly in the program text (an *explicit* declaration), or may be supposed to occur at a given place in the text as a consequence of the semantics of another construct (an *implicit* declaration). 5

Discussion: An implicit declaration generally declares a predefined or inherited operation associated with the definition of a type. This term is used primarily when allowing explicit declarations to override implicit declarations, as part of a type declaration. 5.a

Term entry: declaration — language construct that associates a name with (a view of) an entity 5.b/5
Note: A declaration can appear explicitly in the program text (an explicit declaration), or can be supposed to occur at a given place in the text as a consequence of the semantics of another construct (an implicit declaration).

{*AI95-00318-02*} {*AI05-0255-1*} {*AI05-0277-1*} {*AI12-0061-1*} {*AI12-0308-1*} Each of the following is defined to be a declaration: any basic_declaration; an enumeration_literal_specification; a discriminant_specification; a component_declaration; a defining_identifier of an iterated_component_association; a loop_parameter_specification; a defining_identifier of a chunk_specification; an iterator_specification; a defining_identifier of an iterator_parameter_specification; a parameter_specification; a subprogram_body; an extended_return_object_declaration; an entry_declaration; an entry_index_specification; a choice_parameter_specification; a generic_formal_parameter_declaration. 6/5

Discussion: This list (when basic_declaration is expanded out) contains all syntactic categories that end in "_declaration" or "_specification", except for program unit _specifications. Moreover, it contains subprogram_body. A subprogram_body is a declaration, whether or not it completes a previous declaration. This is a bit strange, 6.a

subprogram_body is not part of the syntax of basic_declaration or library_unit_declaration. A renaming-as-body is considered a declaration. An accept_statement is not considered a declaration. Completions are sometimes declarations, and sometimes not.

7 All declarations contain a *definition* for a *view* of an entity. A view consists of an identification of the entity (the entity *of* the view), plus view-specific characteristics that affect the use of the entity through that view (such as mode of access to an object, formal parameter names and defaults for a subprogram, or visibility to components of a type). In most cases, a declaration also contains the definition for the entity itself (a renaming_declaration is an example of a declaration that does not define a new entity, but instead defines a view of an existing entity (see 8.5)).

7.a.1/5 **Term entry: view of an entity** — representation of an entity that reveals some or all of the properties of the entity
Note: A single entity can have multiple views.

7.a **Discussion:** Most declarations define a view (of some entity) whose view-specific characteristics are unchanging for the life of the view. However, subtypes are somewhat unusual in that they inherit characteristics from whatever view of their type is currently visible. Hence, a subtype is not a *view* of a type; it is more of an indirect reference. By contrast, a private type provides a single, unchanging (partial) view of its full type.

7.1/3 {*AI05-0080-1*} When it is clear from context, the term *object* is used in place of *view of an object*. Similarly, the terms *type* and *subtype* are used in place of *view of a type* and *view of a subtype*, respectively.

7.b/3 **Discussion:** Rules interpreted at compile time generally refer to views of entities, rather than the entities themselves. This is necessary to preserve privacy; characteristics that are not visible should not be used in compile-time rules. Thus, Static Semantics and Legality Rules generally implicitly have "view of". Legality Rules that need to look into the private part are the exception to this interpretation.

7.c/3 On the other hand, run-time rules can work either way, so "view of" should not be assumed in Dynamic Semantics rules.

7.d/5 {*AI12-0191-1*} For example, a reference to the components of an object in a rule that is interpreted at compile time would not apply to components that are not visible. On the other hand, a reference to the components of an object in a dynamic semantics rule would apply to all components of the object, visible or not, including (for tagged objects) components which are not components of the nominal type of the object (see 3.9.1). Other terms, such as "subcomponent" and "part", are interpreted analogously.

8 For each declaration, the language rules define a certain region of text called the *scope* of the declaration (see 8.2). Most declarations associate an identifier with a declared entity. Within its scope, and only there, there are places where it is possible to use the identifier to refer to the declaration, the view it defines, and the associated entity; these places are defined by the visibility rules (see 8.3). At such places the identifier is said to be a *name* of the entity (the direct_name or selector_name); the name is said to *denote* the declaration, the view, and the associated entity (see 8.6). The declaration is said to *declare* the name, the view, and in most cases, the entity itself.

9 As an alternative to an identifier, an enumeration literal can be declared with a character_literal as its name (see 3.5.1), and a function can be declared with an operator_symbol as its name (see 6.1).

10 The syntax rules use the terms defining_identifier, defining_character_literal, and defining_operator_-symbol for the defining occurrence of a name; these are collectively called *defining names*. The terms direct_name and selector_name are used for usage occurrences of identifiers, character_literals, and operator_symbols. These are collectively called *usage names*.

10.a **To be honest:** The terms identifier, character_literal, and operator_symbol are used directly in contexts where the normal visibility rules do not apply (such as the identifier that appears after the **end** of a task_body). Analogous conventions apply to the use of designator, which is the collective term for identifier and operator_symbol.

Dynamic Semantics

11 The process by which a construct achieves its run-time effect is called *execution*. This process is also called *elaboration* for declarations and *evaluation* for expressions. One of the terms execution,

elaboration, or evaluation is defined by this Reference Manual for each construct that has a run-time effect.

Term entry: execution — process by which a construct achieves its run-time effect 11.a.1/5
Note: Execution of a declaration is also called elaboration. Execution of an expression is also called evaluation.

To be honest: The term elaboration is also used for the execution of certain constructs that are not declarations, and the term evaluation is used for the execution of certain constructs that are not expressions. For example, subtype_indications are elaborated, and ranges are evaluated. 11.a

For bodies, execution and elaboration are both explicitly defined. When we refer specifically to the execution of a body, we mean the explicit definition of execution for that kind of body, not its elaboration. 11.b

Discussion: Technically, "the execution of a declaration" and "the elaboration of a declaration" are synonymous. We use the term "elaboration" of a construct when we know the construct is elaborable. When we are talking about more arbitrary constructs, we use the term "execution". For example, we use the term "erroneous execution", to refer to any erroneous execution, including erroneous elaboration or evaluation. 11.c

When we explicitly define evaluation or elaboration for a construct, we are implicitly defining execution of that construct. 11.d

We also use the term "execution" for things like statements, which are executable, but neither elaborable nor evaluable. We considered using the term "execution" only for nonelaborable, nonevaluable constructs, and defining the term "action" to mean what we have defined "execution" to mean. We rejected this idea because we thought three terms that mean the same thing was enough — four would be overkill. Thus, the term "action" is used only informally in the standard (except where it is defined as part of a larger term, such as "protected action"). 11.e

Term entry: elaboration — process by which a declaration achieves its run-time effect 11.e.1/5
Note: Elaboration is one of the forms of execution.

Term entry: evaluation — process by which an expression achieves its run-time effect 11.e.2/5
Note: Evaluation is one of the forms of execution.

To be honest: A construct is *elaborable* if elaboration is defined for it. A construct is *evaluable* if evaluation is defined for it. A construct is *executable* if execution is defined for it. 11.f

Discussion: Don't confuse “elaborable” with “preelaborable” (defined in 10.2.1). 11.g

{*AI95-00114-01*} Evaluation of an evaluable construct produces a result that is either a value, a denotation, or a range. The following are evaluable: expression; name prefix; range; entry_index_specification; and possibly discrete_range. The last one is curious — RM83 uses the term “evaluation of a discrete_range”, but never defines it. One might presume that the evaluation of a discrete_range consists of the evaluation of the range or the subtype_indication, depending on what it is. But subtype_indications are not evaluated; they are elaborated. 11.h/2

Intuitively, an *executable* construct is one that has a defined run-time effect (which may be null). Since execution includes elaboration and evaluation as special cases, all elaborable and all evaluable constructs are also executable. Hence, most constructs in Ada are executable. An important exception is that the constructs inside a generic unit are not executable directly, but rather are used as a template for (generally) executable constructs in instances of the generic. 11.i

NOTE At compile time, the declaration of an entity *declares* the entity. At run time, the elaboration of the declaration *creates* the entity. 12

Ramification: Syntactic categories for declarations are named either *entity*_declaration (if they include a trailing semicolon) or *entity*_specification (if not). 12.a

The various kinds of named entities that can be declared are as follows: an object (including components and parameters), a named number, a type (the name always refers to its first subtype), a subtype, a subprogram (including enumeration literals and operators), a single entry, an entry family, a package, a protected or task unit (which corresponds to either a type or a single object), an exception, a generic unit, a label, and the name of a statement. 12.b

Identifiers are also associated with names of pragmas, arguments to pragmas, and with attributes, but these are not user-definable. 12.c

Wording Changes from Ada 83

The syntax rule for defining_identifier is new. It is used for the defining occurrence of an identifier. Usage occurrences use the direct_name or selector_name syntactic categories. Each occurrence of an identifier (or simple_name), character_literal, or operator_symbol in the Ada 83 syntax rules is handled as follows in Ada 95: 12.d

- It becomes a defining_identifier, defining_character_literal, or defining_operator_symbol (or some syntactic category composed of these), to indicate a defining occurrence; 12.e

12.f
- It becomes a direct_name, in usage occurrences where the usage is required (in Clause 8 to be directly visible;

12.g
- It becomes a selector_name, in usage occurrences where the usage is required (in Clause 8) to be visible but not necessarily directly visible;

12.h
- It remains an identifier, character_literal, or operator_symbol, in cases where the visibility rules do not apply (such as the designator that appears after the **end** of a subprogram_body).

12.i For declarations that come in “two parts” (program unit declaration plus body, private or incomplete type plus full type, deferred constant plus full constant), we consider both to be defining occurrences. Thus, for example, the syntax for package_body uses defining_identifier after the reserved word **body**, as opposed to direct_name.

12.j The defining occurrence of a statement name is in its implicit declaration, not where it appears in the program text. Considering the statement name itself to be the defining occurrence would complicate the visibility rules.

12.k The phrase “visible by selection” is not used in Ada 95. It is subsumed by simply “visible” and the Name Resolution Rules for selector_names.

12.l/3 {*AI05-0299-1*} (Note that in Ada 95, a declaration is visible at all places where one could have used a selector_name, not just at places where a selector_name was actually used. Thus, the places where a declaration is directly visible are a subset of the places where it is visible. See Clause 8 for details.)

12.m We use the term “declaration” to cover _specifications that declare (views of) objects, such as parameter_specifications. In Ada 83, these are referred to as a “form of declaration”, but it is not entirely clear that they are considered simply “declarations”.

12.n/3 {*AI05-0299-1*} RM83 contains an incomplete definition of "elaborated" in this subclause: it defines "elaborated" for declarations, declarative_parts, declarative_items and compilation_units, but "elaboration" is defined elsewhere for various other constructs. To make matters worse, Ada 95 has a different set of elaborable constructs. Instead of correcting the list, it is more maintainable to refer to the term "elaborable," which is defined in a distributed manner.

12.o RM83 uses the term “has no other effect” to describe an elaboration that doesn't do anything except change the state from not-yet-elaborated to elaborated. This was a confusing wording, because the answer to “other than what?” was to be found many pages away. In Ada 95, we change this wording to “has no effect” (for things that truly do nothing at run time), and “has no effect other than to establish that so-and-so can happen without failing the Elaboration_Check” (for things where it matters).

12.p We make it clearer that the term "execution" covers elaboration and evaluation as special cases. This was implied in RM83. For example, "erroneous execution" can include any execution, and RM83-9.4(3) has, "The task designated by any other task object depends on the master whose execution creates the task object;" the elaboration of the master's declarative_part is doing the task creation.

Wording Changes from Ada 95

12.q/2 {*AI95-00318-02*} Added extended_return_statement to the list of declarations.

12.r/2 {*AI95-00348-01*} Added null procedures (see 6.7) to the syntax.

Wording Changes from Ada 2005

12.s/3 {*AI05-0177-1*} Added expression functions (see 6.8) to the syntax.

3.2 Types and Subtypes

Static Semantics

1 A *type* is characterized by a set of values, and a set of *primitive operations* which implement the fundamental aspects of its semantics. An *object* of a given type is a run-time entity that contains (has) a value of the type.

1.a/5 **Term entry: type** — defining characteristic of each object and expression of the language, with an associated set of values, and a set of primitive operations that implement the fundamental aspects of its semantics
Note: Types are grouped into categories. Most language-defined categories of types are also classes of types.

1.b/5 **Term entry: subtype** — type together with optional constraints, null exclusions, and predicates, which constrain the values of the type to the subset that satisfies the implied conditions

2/5 {*AI95-00442-01*} {*AI12-0451-1*} Types are grouped into *categories* of types. There exist several *language-defined categories* of types (summarized in the NOTE below), reflecting the similarity of their

values and primitive operations. [Most categories of types form *classes* of types.] *Elementary* types are those whose values are logically indivisible; *composite* types are those whose values are composed of *component* values.

Proof: {*AI95-00442-01*} The formal definition of *category* and *class* is found in 3.4. 2.a/2

Term entry: class of types — set of types that is closed under derivation, which means that if a given type is in the class, then all types derived from that type are also in the class 2.b/5
Note: The set of types of a class share common properties, such as their primitive operations.

Term entry: category of types — set of types with one or more common properties, such as primitive operations 2.c/5
Note: A category of types that is closed under derivation is also known as a class.

Term entry: elementary type — type that does not have components 2.d/5

Term entry: composite type — type with components, such as an array or record 2.e/5

Term entry: scalar type — either a discrete type or a real type 2.f/5

Term entry: access type — type that has values that designate aliased objects 2.g/5
Note: Access types correspond to "pointer types" or "reference types" in some other languages.

Term entry: discrete type — type that is either an integer type or an enumeration type 2.h/5

Term entry: real type — type that has values that are approximations of the real numbers 2.i/5
Note: Floating point and fixed point types are real types.

Term entry: integer type — type that represents signed or modular integers 2.j/5
Note: A signed integer type has a base range that includes both positive and negative numbers, and has operations that can raise an exception when the result is outside the base range. A modular type has a base range whose lower bound is zero, and has operations with "wraparound" semantics. Modular types subsume what are called "unsigned types" in some other languages.

Term entry: enumeration type — type defined by an enumeration of its values, which can be denoted by identifiers or character literals 2.k/5

Term entry: character type — enumeration type whose values include characters 2.l/5

Term entry: record type — composite type consisting of zero or more named components, possibly of different types 2.m/5

Term entry: record extension — type that extends another type optionally with additional components 2.n/5

Term entry: array type — composite type whose components are all of the same type 2.o/5

Term entry: task type — composite type used to represent active entities which execute concurrently and that can communicate via queued task entries 2.p/5
Note: The top-level task of a partition is called the environment task.

Term entry: protected type — composite type whose components are accessible only through one of its protected operations, which synchronize concurrent access by multiple tasks 2.q/5

Term entry: partial view — view of a type that reveals only some of its properties 2.r/5
Note: The remaining properties are defined by the full view given elsewhere.

Term entry: full view — view of a type that reveals all of its properties 2.s/5
Note: There can be other views of the type that reveal fewer properties.

Term entry: incomplete view — view of a type that reveals minimal properties 2.t/5
Note: The remaining properties are defined by the full view given elsewhere.

Term entry: private type — type that defines a partial view 2.u/5
Note: Private types can be used for defining abstractions that hide unnecessary details from their clients.

Term entry: private extension — type that extends another type, with the additional properties hidden from its clients 2.v/5

Term entry: full type — type that defines a full view 2.w/5

Term entry: incomplete type — type that defines an incomplete view 2.x/5
Note: Incomplete types can be used for defining recursive data structures.

The elementary types are the *scalar* types (*discrete* and *real*) and the *access* types (whose values provide access to objects or subprograms). Discrete types are either *integer* types or are defined by enumeration of their values (*enumeration* types). Real types are either *floating point* types or *fixed point* types. 3

{*AI95-00251-01*} {*AI95-00326-01*} The composite types are the *record* types, *record extensions*, *array* types, *interface* types, *task* types, and *protected* types. 4/2

4.a/2 *This paragraph was deleted.*{*AI95-00442-01*}

4.1/2 {*AI95-00326-01*} There can be multiple views of a type with varying sets of operations. [An *incomplete* type represents an incomplete view (see 3.10.1) of a type with a very restricted usage, providing support for recursive data structures. A *private* type or *private extension* represents a partial view (see 7.3) of a type, providing support for data abstraction. The full view (see 3.2.1) of a type represents its complete definition.] An incomplete or partial view is considered a composite type[, even if the full view is not].

4.b/3 **Proof:** {*AI05-0299-1*} The real definitions of the views are in the referenced subclauses.

5/2 {*AI95-00326-01*} Certain composite types (and views thereof) have special components called *discriminants* whose values affect the presence, constraints, or initialization of other components. Discriminants can be thought of as parameters of the type.

6/2 {*AI95-00366-01*} The term *subcomponent* is used in this Reference Manual in place of the term component to indicate either a component, or a component of another subcomponent. Where other subcomponents are excluded, the term component is used instead. Similarly, a *part* of an object or value is used to mean the whole object or value, or any set of its subcomponents. The terms component, subcomponent, and part are also applied to a type meaning the component, subcomponent, or part of objects and values of the type.

6.a **Discussion:** The definition of “part” here is designed to simplify rules elsewhere. By design, the intuitive meaning of “part” will convey the correct result to the casual reader, while this formalistic definition will answer the concern of the compiler-writer.

6.b We use the term “part” when talking about the parent part, ancestor part, or extension part of a type extension. In contexts such as these, the part might represent an empty set of subcomponents (e.g. in a null record extension, or a nonnull extension of a null record). We also use “part” when specifying rules such as those that apply to an object with a “controlled part” meaning that it applies if the object as a whole is controlled, or any subcomponent is.

7/2 {*AI95-00231-01*} The set of possible values for an object of a given type can be subjected to a condition that is called a *constraint* (the case of a *null constraint* that specifies no restriction is also included)[; the rules for which values satisfy a given kind of constraint are given in 3.5 for range_constraints, 3.6.1 for index_constraints, and 3.7.1 for discriminant_constraints]. The set of possible values for an object of an access type can also be subjected to a condition that excludes the null value (see 3.10).

7.a/5 **Ramification:** {*AI12-0140-1*} “Null constraint” includes the cases of no explicit constraint, as well as unknown discriminants and unconstrained array type declarations (which are explicit ways to declare no constraint).

8/5 {*AI95-00231-01*} {*AI95-00415-01*} {*AI12-0445-1*} A *subtype* of a given type is a combination of the type, a constraint on values of the type, and certain attributes specific to the subtype. The given type is called the *type of the subtype*. Similarly, the associated constraint is called the *constraint of the subtype*. The set of values of a subtype consists of the values of its type that satisfy its constraint and any exclusion of the null value. Such values *belong* to the subtype. The other values of the type are *outside* the subtype.

8.a **Discussion:** We make a strong distinction between a type and its subtypes. In particular, a type is *not* a subtype of itself. There is no constraint associated with a type (not even a null one), and type-related attributes are distinct from subtype-specific attributes.

8.b **Discussion:** We no longer use the term "base type." All types were "base types" anyway in Ada 83, so the term was redundant, and occasionally confusing. In the RM95 we say simply "the type *of* the subtype" instead of "the base type of the subtype."

8.c **Ramification:** The value subset for a subtype might be empty, and need not be a proper subset.

8.d/2 **To be honest:** {*AI95-00442-01*} Any name of a category of types (such as “discrete”, “real”, or “limited”) is also used to qualify its subtypes, as well as its objects, values, declarations, and definitions, such as an “integer type declaration” or an “integer value”. In addition, if a term such as “parent subtype” or “index subtype” is defined, then the corresponding term for the type of the subtype is “parent type” or “index type”.

8.e **Discussion:** We use these corresponding terms without explicitly defining them, when the meaning is obvious.

A subtype is called an *unconstrained* subtype if its type has unknown discriminants, or if its type allows range, index, or discriminant constraints, but the subtype does not impose such a constraint; otherwise, the subtype is called a *constrained* subtype (since it has no unconstrained characteristics). 9

Discussion: In an earlier version of Ada 9X, "constrained" meant "has a nonnull constraint." However, we changed to this definition since we kept having to special case composite non-array/nondiscriminated types. It also corresponds better to the (now obsolescent) attribute 'Constrained. 9.a

For scalar types, “constrained” means “has a nonnull constraint”. For composite types, in implementation terms, “constrained” means that the size of all objects of the subtype is the same, assuming a typical implementation model. 9.b

Class-wide subtypes are always unconstrained. 9.c

NOTE {*AI95-00442-01*} Any set of types can be called a “category” of types, and any set of types that is closed under derivation (see 3.4) can be called a “class” of types. However, only certain categories and classes are used in the description of the rules of the language — generally those that have their own particular set of primitive operations (see 3.2.3), or that correspond to a set of types that are matched by a given kind of generic formal type (see 12.5). The following are examples of “interesting” *language-defined classes*: elementary, scalar, discrete, enumeration, character, boolean, integer, signed integer, modular, real, floating point, fixed point, ordinary fixed point, decimal fixed point, numeric, access, access-to-object, access-to-subprogram, composite, array, string, (untagged) record, tagged, task, protected, nonlimited. Special syntax is provided to define types in each of these classes. In addition to these classes, the following are examples of “interesting” *language-defined categories*: abstract, incomplete, interface, limited, private, record. 10/2

Discussion: A *value* is a run-time entity with a given type which can be assigned to an object of an appropriate subtype of the type. An *operation* is a program entity that operates on zero or more operands to produce an effect, or yield a result, or both. 10.a

Ramification: {*AI95-00442-01*} Note that a type's category (and class) depends on the place of the reference — a private type is composite outside and possibly elementary inside. It's really the *view* that is elementary or composite. Note that although private types are composite, there are some properties that depend on the corresponding full view — for example, parameter passing modes, and the constraint checks that apply in various places. 10.b/2

{*AI95-00345-01*} {*AI95-00442-01*} Every property of types forms a category, but not every property of types represents a class. For example, the set of all abstract types does not form a class, because this set is not closed under derivation. Similarly, the set of all interface types does not form a class. 10.c/2

{*AI95-00442-01*} The set of limited types does not form a class (since nonlimited types can inherit from limited interfaces), but the set of nonlimited types does. The set of tagged record types and the set of tagged private types do not form a class (because each of them can be extended to create a type of the other category); that implies that the set of record types and the set of private types also do not form a class (even though untagged record types and untagged private types do form a class). In all of these cases, we can talk about the category of the type; for instance, we can talk about the “category of limited types”. 10.d/2

{*AI95-00442-01*} Normatively, the *language-defined classes* are those that are defined to be inherited on derivation by 3.4; other properties either aren't interesting or form categories, not classes. 10.e/2

{*AI95-00442-01*} These language-defined categories are organized like this: 11/2

{*AI95-00345-01*} all types 12/2

- elementary
  - scalar
    - discrete
      - enumeration
        - character
        - boolean
        - other enumeration
      - integer
        - signed integer
        - modular integer
    - real
      - floating point
      - fixed point
        - ordinary fixed point
        - decimal fixed point
  - access
    - access-to-object
    - access-to-subprogram
- composite
  - untagged

array
string
other array
record
task
protected
tagged (including interfaces)
nonlimited tagged record
limited tagged
limited tagged record
synchronized tagged
tagged task
tagged protected

13/2 {*AI95-00345-01*} {*AI95-00442-01*} There are other categories, such as “numeric” and “discriminated”, which represent other categorization dimensions, but do not fit into the above strictly hierarchical picture.

13.a.1/2 **Discussion:** {*AI95-00345-01*} {*AI95-00442-01*} Note that this is also true for some categories mentioned in the chart. The category “task” includes both untagged tasks and tagged tasks. Similarly for “protected”, “limited”, and “nonlimited” (note that limited and nonlimited are not shown for untagged composite types).

Wording Changes from Ada 83

13.a/3 {*AI05-0299-1*} This subclause now precedes the subclauses on objects and named numbers, to cut down on the number of forward references.

13.b We have dropped the term "base type" in favor of simply "type" (all types in Ada 83 were "base types" so it wasn't clear when it was appropriate/necessary to say "base type"). Given a subtype S of a type T, we call T the "type of the subtype S."

Wording Changes from Ada 95

13.c/2 {*AI95-00231-01*} Added a mention of null exclusions when we're talking about constraints (these are not constraints, but they are similar).

13.d/2 {*AI95-00251-01*} Defined an interface type to be a composite type.

13.e/2 {*AI95-00326-01*} Revised the wording so that it is clear that an incomplete view is similar to a partial view in terms of the language.

13.f/2 {*AI95-00366-01*} Added a definition of component of a type, subcomponent of a type, and part of a type. These are commonly used in the standard, but they were not previously defined.

13.g/3 {*AI95-00442-01*} {*AI05-0299-1*} Reworded most of this subclause to use category rather than class, since so many interesting properties are not, strictly speaking, classes. Moreover, there was no normative description of exactly which properties formed classes, and which did not. The real definition of class, along with a list of properties, is now in 3.4.

3.2.1 Type Declarations

1 A type_declaration declares a type and its first subtype.

Syntax

2 type_declaration ::= full_type_declaration
| incomplete_type_declaration
| private_type_declaration
| private_extension_declaration

3/3 {*AI05-0183-1*} full_type_declaration ::=
type defining_identifier [known_discriminant_part] **is** type_definition
[aspect_specification];
| task_type_declaration
| protected_type_declaration

4/2 {*AI95-00251-01*} type_definition ::=
enumeration_type_definition | integer_type_definition

| real_type_definition | array_type_definition
| record_type_definition | access_type_definition
| derived_type_definition | interface_type_definition

Legality Rules

A given type shall not have a subcomponent whose type is the given type itself. 5

Static Semantics

{*AI12-0449-1*} The defining_identifier of a type_declaration denotes the *first subtype* of the type. The known_discriminant_part, if any, defines the discriminants of the type (see 3.7). The remainder of the type_declaration defines the remaining characteristics of (the view of) the type. 6/5

{*AI95-00230-01*} A type defined by a type_declaration is a *named* type; such a type has one or more nameable subtypes. Certain other forms of declaration also include type definitions as part of the declaration for an object. The type defined by such a declaration is *anonymous* — it has no nameable subtypes. For explanatory purposes, this document sometimes refers to an anonymous type by a pseudo-name, written in italics, and uses such pseudo-names at places where the syntax normally requires an identifier. For a named type whose first subtype is T, this document sometimes refers to the type of T as simply "the type T". 7/2

Ramification: {*AI95-00230-01*} The only user-defined types that can be anonymous in the above sense are array, access, task, and protected types. An anonymous array, task, or protected type can be defined as part of an object_declaration. An anonymous access type can be defined as part of numerous other constructs. 7.a/2

{*AI95-00230-01*} {*AI95-00326-01*} A named type that is declared by a full_type_declaration, or an anonymous type that is defined by an access_definition or as part of declaring an object of the type, is called a *full type*. The declaration of a full type also declares the *full view* of the type. The type_definition, task_definition, protected_definition, or access_definition that defines a full type is called a *full type definition*. [Types declared by other forms of type_declaration are not separate types; they are partial or incomplete views of some full type.] 8/2

To be honest: Class-wide, universal, and root numeric types are full types. 8.a

Reason: {*AI95-00230-01*} We need to mention access_definition separately, as it may occur in renames, which do not declare objects. 8.b/2

{*AI12-0449-1*} The definition of a type implicitly declares certain *predefined operators* that operate on the type, according to what classes the type belongs, as specified in 4.5. 9/5

Discussion: We no longer talk about the implicit declaration of basic operations. These are treated like an if_statement — they don't need to be declared, but are still applicable to only certain classes of types. 9.a

The *predefined types* [(for example the types Boolean, Wide_Character, Integer, *root_integer*, and *universal_integer*)] are the types that are defined in [a predefined library package called] Standard[; this package also includes the [(implicit)] declarations of their predefined operators]. [The package Standard is described in A.1.] 10

Ramification: We use the term "predefined" to refer to entities declared in the visible part of Standard, to implicitly declared operators of a type whose semantics are defined by the language, to Standard itself, and to the "predefined environment". We do not use this term to refer to library packages other than Standard. For example Text_IO is a language-defined package, not a predefined package, and Text_IO.Put_Line is not a predefined operation. 10.a

Dynamic Semantics

The elaboration of a full_type_declaration consists of the elaboration of the full type definition. Each elaboration of a full type definition creates a distinct type and its first subtype. 11

11.a **Reason:** The creation is associated with the type *definition*, rather than the type *declaration*, because there are types that are created by full type definitions that are not immediately contained within a type declaration (e.g. an array object declaration, a singleton task declaration, etc.).

11.b **Ramification:** Any implicit declarations that occur immediately following the full type definition are elaborated where they (implicitly) occur.

Examples

12 *Examples of type definitions:*

13
```
(White, Red, Yellow, Green, Blue, Brown, Black)
range 1 .. 72
array(1 .. 10) of Integer
```

14 *Examples of type declarations:*

15
```
type Color  is (White, Red, Yellow, Green, Blue, Brown, Black);
type Column is range 1 .. 72;
type Table  is array(1 .. 10) of Integer;
```

16 NOTE Each of the above examples declares a named type. The identifier given denotes the first subtype of the type. Other named subtypes of the type can be declared with subtype_declarations (see 3.2.2). Although names do not directly denote types, a phrase like “the type Column” is sometimes used in this document to refer to the type of Column, where Column denotes the first subtype of the type. For an example of the definition of an anonymous type, see the declaration of the array Color_Table in 3.3.1; its type is anonymous — it has no nameable subtypes.

Wording Changes from Ada 83

16.a The syntactic category full_type_declaration now includes task and protected type declarations.

16.b We have generalized the concept of first-named subtype (now called simply “first subtype”) to cover all kinds of types, for uniformity of description elsewhere. RM83 defined first-named subtype in Section 13. We define first subtype here, because it is now a more fundamental concept. We renamed the term, because in Ada 95 some first subtypes have no name.

16.c/2 {*AI95-00230-01*} We no longer elaborate discriminant_parts, because there is nothing to do, and it was complex to say that you only wanted to elaborate it once for a private or incomplete type. This is also consistent with the fact that subprogram specifications are not elaborated (neither in Ada 83 nor in Ada 95). Note, however, that an access_definition appearing in a discriminant_part is elaborated at the full_type_declaration (for a nonlimited type) or when an object with such a discriminant is created (for a limited type).

Wording Changes from Ada 95

16.d/2 {*AI95-00230-01*} Added wording so that anonymous access types are always full types, even if they appear in renames.

16.e/2 {*AI95-00251-01*} Added interface types (see 3.9.4) to the syntax.

16.f/2 {*AI95-00326-01*} Added a definition of full view, so that all types have a well-defined full view.

Extensions to Ada 2005

16.g/3 {*AI05-0183-1*} An optional aspect_specification can be used in a full_type_declaration. This is described in 13.1.1.

3.2.2 Subtype Declarations

1 A subtype_declaration declares a subtype of some previously declared type, as defined by a subtype_indication.

Syntax

2/3 {*AI05-0183-1*} subtype_declaration ::=
subtype defining_identifier **is** subtype_indication
[aspect_specification];

3/2 {*AI95-00231-01*} subtype_indication ::= [null_exclusion] subtype_mark [constraint]

4 subtype_mark ::= *subtype*_name

Ramification: Note that name includes attribute_reference; thus, S'Base can be used as a subtype_mark. 4.a

Reason: We considered changing subtype_mark to subtype_name. However, existing users are used to the word "mark," so we're keeping it. 4.b

constraint ::= scalar_constraint | composite_constraint 5

scalar_constraint ::= 6
range_constraint | digits_constraint | delta_constraint

composite_constraint ::= 7
index_constraint | discriminant_constraint

Name Resolution Rules

A subtype_mark shall resolve to denote a subtype. The type *determined by* a subtype_mark is the type of the subtype denoted by the subtype_mark. 8

Ramification: {*AI05-0005-1*} Types are never directly named; all subtype_marks denote subtypes — possibly an unconstrained (base) subtype, but never the type. When we use the term *anonymous type* we really mean a type with no nameable subtypes. 8.a/3

Dynamic Semantics

The elaboration of a subtype_declaration consists of the elaboration of the subtype_indication. The elaboration of a subtype_indication creates a new subtype. If the subtype_indication does not include a constraint, the new subtype has the same (possibly null) constraint as that denoted by the subtype_mark. The elaboration of a subtype_indication that includes a constraint proceeds as follows: 9

- The constraint is first elaborated. 10
- A check is then made that the constraint is *compatible* with the subtype denoted by the subtype_mark. 11

Ramification: The checks associated with constraint compatibility are all Range_Checks. Discriminant_Checks and Index_Checks are associated only with checks that a value satisfies a constraint. 11.a

The condition imposed by a constraint is the condition obtained after elaboration of the constraint. The rules defining compatibility are given for each form of constraint in the appropriate subclause. These rules are such that if a constraint is *compatible* with a subtype, then the condition imposed by the constraint cannot contradict any condition already imposed by the subtype on its values. The exception Constraint_Error is raised if any check of compatibility fails. 12

To be honest: The condition imposed by a constraint is named after it — a range_constraint imposes a range constraint, etc. 12.a

Ramification: A range_constraint causes freezing of its type. Other constraints do not. 12.b

NOTE {*AI12-0440-1*} A scalar_constraint can be applied to a subtype of an appropriate scalar type (see 3.5, 3.5.9, and J.3), even if the subtype is already constrained. On the other hand, a composite_constraint can be applied to a composite subtype (or an access-to-composite subtype) only if the composite subtype is unconstrained (see 3.6.1 and 3.7.1). 13/5

Examples

Examples of subtype declarations: 14

```
{AI95-00433-01} subtype Rainbow   is Color range Red .. Blue;       -- see 3.2.1
subtype Red_Blue  is Rainbow;
subtype Int       is Integer;
subtype Small_Int is Integer range -10 .. 10;
subtype Up_To_K   is Column range 1 .. K;           -- see 3.2.1
subtype Square    is Matrix(1 .. 10, 1 .. 10);      -- see 3.6
subtype Male      is Person(Sex => M);              -- see 3.10.1
subtype Binop_Ref is not null Binop_Ptr;            -- see 3.10
```
15/2

Incompatibilities With Ada 83

15.a In Ada 95, all range_constraints cause freezing of their type. Hence, a type-related representation item for a scalar type has to precede any range_constraints whose type is the scalar type.

Wording Changes from Ada 83

15.b Subtype_marks allow only subtype names now, since types are never directly named. There is no need for RM83-3.3.2(3), which says a subtype_mark can denote both the type and the subtype; in Ada 95, you denote an unconstrained (base) subtype if you want, but never the type.

15.c The syntactic category type_mark is now called subtype_mark, since it always denotes a subtype.

Extensions to Ada 95

15.d/2 {*AI95-00231-01*} An optional null_exclusion can be used in a subtype_indication. This is described in 3.10.

Extensions to Ada 2005

15.e/3 {*AI05-0183-1*} An optional aspect_specification can be used in a subtype_declaration. This is described in 13.1.1.

3.2.3 Classification of Operations

Static Semantics

1/2 {*AI95-00416-01*} An operation *operates on a type T* if it yields a value of type *T*, if it has an operand whose expected type (see 8.6) is *T*, or if it has an access parameter or access result type (see 6.1) designating *T*. A predefined operator, or other language-defined operation such as assignment or a membership test, that operates on a type, is called a *predefined operation* of the type. The *primitive operations* of a type are the predefined operations of the type, plus any user-defined primitive subprograms.

1.a.1/5 **Term entry: primitive operations of a type** — operations (such as subprograms) declared together with the type declarations
Note: Primitive operations are inherited by other types in the same derivation class of types.

1.a **To be honest:** Protected subprograms are not considered to be “primitive subprograms”, even though they are subprograms, and they are inherited by derived types.

1.b **Discussion:** We use the term “primitive subprogram” in most of the rest of the manual. The term “primitive operation” is used mostly in conceptual discussions.

2 The *primitive subprograms* of a specific type are defined as follows:

3 • The predefined operators of the type (see 4.5);

4 • For a derived type, the inherited (see 3.4) user-defined subprograms;

5 • For an enumeration type, the enumeration literals (which are considered parameterless functions — see 3.5.1);

6 • For a specific type declared immediately within a package_specification, any subprograms (in addition to the enumeration literals) that are explicitly declared immediately within the same package_specification and that operate on the type;

6.1/3 • {*AI05-0128-1*} For a specific type with an explicitly declared primitive "=" operator whose result type is Boolean, the corresponding "/=" operator (see 6.6);

7/2 • {*AI95-00200-01*} For a nonformal type, any subprograms not covered above [that are explicitly declared immediately within the same declarative region as the type] and that override (see 8.3) other implicitly declared primitive subprograms of the type.

7.a **Discussion:** In Ada 83, only subprograms declared in the visible part were “primitive” (i.e. derivable). In Ada 95, mostly because of child library units, we include all operations declared in the private part as well, and all operations that override implicit declarations.

Ramification: It is possible for a subprogram to be primitive for more than one type, though it is illegal for a subprogram to be primitive for more than one tagged type. See 3.9. 7.b

Discussion: The order of the implicit declarations when there are both predefined operators and inherited subprograms is described in 3.4, “Derived Types and Classes”. 7.c

Ramification: {*AI95-00200-01*} Subprograms declared in a generic package specification are never primitive for a formal type, even if they happen to override an operation of the formal type. This includes formal subprograms, which are never primitive operations (that's true even for an abstract formal subprogram). 7.d/2

A primitive subprogram whose designator is an operator_symbol is called a *primitive operator*. 8

Incompatibilities With Ada 83

The attribute S'Base is no longer defined for nonscalar subtypes. Since this was only permitted as the prefix of another attribute, and there are no interesting nonscalar attributes defined for an unconstrained composite or access subtype, this should not affect any existing programs. 8.a

Extensions to Ada 83

The primitive subprograms (derivable subprograms) include subprograms declared in the private part of a package specification as well, and those that override implicitly declared subprograms, even if declared in a body. 8.b

Wording Changes from Ada 83

We have dropped the confusing term *operation of a type* in favor of the more useful *primitive operation of a type* and the phrase *operates on a type*. 8.c

The description of S'Base has been moved to 3.5, “Scalar Types” because it is now defined only for scalar types. 8.d

Wording Changes from Ada 95

{*AI95-00200-01*} Clarified that a formal subprogram that happens to override a primitive operation of a formal type is not a primitive operation (and thus not a dispatching operation) of the formal type. 8.e/2

{*AI95-00416-01*} Added wording to include access result types in the kinds of operations that operate on a type T. 8.f/2

Wording Changes from Ada 2005

{*AI05-0128-1*} **Correction:** The implicitly declared "/=" for a primitive "=" operator is also primitive; this makes it eligible to be made visible by a **use type** clause. 8.g/3

3.2.4 Subtype Predicates

{*AI05-0153-3*} {*AI05-0269-1*} {*AI05-0299-1*} {*AI12-0396-1*} {*AI12-0419-1*} The language-defined *predicate aspects* Static_Predicate and Dynamic_Predicate may be used to define properties of subtypes. A *predicate specification* is an aspect_specification for one of the two predicate aspects. General rules for aspects and aspect_specifications are found in Clause 13 (13.1 and 13.1.1 respectively). The predicate aspects are assertion aspects (see 11.4.2). [The predicate aspects are not inherited, but their effects are additive, as defined below.] 1/5

Aspect Description for Static_Predicate: Condition that will hold true for objects of a given subtype; the subtype may be static. 1.a/5

Aspect Description for Dynamic_Predicate: Condition that will hold true for objects of a given subtype; the subtype is not static. 1.b/5

Name Resolution Rules

{*AI05-0153-3*} The expected type for a predicate aspect expression is any boolean type. 2/3

Static Semantics

{*AI05-0153-3*} A predicate specification may be given on a type_declaration or a subtype_declaration, and applies to the declared subtype. In addition, predicate specifications apply to certain other subtypes: 3/3

4/4 • {*AI12-0071-1*} {*AI12-0099-1*} For a (first) subtype defined by a type declaration, any predicates of parent or progenitor subtypes apply.

5/3 • For a subtype created by a subtype_indication, the predicate of the subtype denoted by the subtype_mark applies.

6/4 *This paragraph was deleted.*{*AI05-0153-3*} {*AI12-0071-1*}

7/3 {*AI05-0290-1*} Predicate checks are defined to be *enabled* or *disabled* for a given subtype as follows:

8/3 • If a subtype is declared by a type_declaration or subtype_declaration that includes a predicate specification, then:

9/3 • if performing checks is required by the Static_Predicate assertion policy (see 11.4.2) and the declaration includes a Static_Predicate specification, then predicate checks are enabled for the subtype;

10/3 • if performing checks is required by the Dynamic_Predicate assertion policy (see 11.4.2) and the declaration includes a Dynamic_Predicate specification, then predicate checks are enabled for the subtype;

11/3 • otherwise, predicate checks are disabled for the subtype[, regardless of whether predicate checking is enabled for any other subtypes mentioned in the declaration];

12/4 • {*AI12-0099-1*} If a subtype is defined by a type declaration that does not include a predicate specification, then predicate checks are enabled for the subtype if and only if any predicate checks are enabled for parent or progenitor subtypes;

13/3 • If a subtype is created by a subtype_indication other than in one of the previous cases, then predicate checks are enabled for the subtype if and only if predicate checks are enabled for the subtype denoted by the subtype_mark;

14/3 • Otherwise, predicate checks are disabled for the given subtype.

14.a/3 **Discussion:** In this case, no predicate specifications can apply to the subtype and so it doesn't typically matter whether predicate checks are enabled. This rule does make a difference, however, when determining whether predicate checks are enabled for another type when this type is one of multiple progenitors. See the “derived type declaration” wording above.

14.b/4 {*AI12-0071-1*} Even when predicate checks are disabled, a predicate can affect various Legality Rules, the results of membership tests, the items in a **for** loop, and the result of the Valid attribute.

14.1/4 {*AI12-0054-2*} For a subtype with a directly-specified predicate aspect, the following additional language-defined aspect may be specified with an aspect_specification (see 13.1.1):

14.2/4 Predicate_Failure
: This aspect shall be specified by an expression, which determines the action to be performed when a predicate check fails because a directly-specified predicate aspect of the subtype evaluates to False, as explained below.

14.b.1/4 **Aspect Description for Predicate_Failure:** Action to be performed when a predicate check fails.

Name Resolution Rules

14.3/4 {*AI12-0054-2*} The expected type for the Predicate_Failure expression is String.

Legality Rules

15/3 {*AI05-0153-3*} {*AI05-0269-1*} The expression of a Static_Predicate specification shall be *predicate-static*; that is, one of the following:

16/3 • a static expression;

17/4 • {*AI12-0039-1*} a membership test whose *tested*_simple_expression is the current instance, and whose membership_choice_list meets the requirements for a static membership test (see 4.9);

- a case_expression whose *selecting*_expression is the current instance, and whose *dependent*_expressions are static expressions; 18/3
- a call to a predefined equality or ordering operator, where one operand is the current instance, and the other is a static expression; 19/3
- {*AI05-0262-1*} {*AI12-0099-1*} a call to a predefined boolean operator **and**, **or**, **xor**, or **not**, where each operand is predicate-static; 20/4
- {*AI05-0269-1*} a short-circuit control form where both operands are predicate-static; or 21/3
- a parenthesized predicate-static expression. 22/3

{*AI05-0262-1*} A predicate shall not be specified for an incomplete subtype. 23/3

Reason: The expression of such a predicate could not depend on the properties of the value of the type (since it doesn't have any), so it is useless and we don't want to require the added complexity needed to support it. 23.a/3

{*AI05-0287-1*} If a predicate applies to a subtype, then that predicate shall not mention any other subtype to which the same predicate applies. 24/3

Reason: This is intended to prevent recursive predicates, which cause definitional problems for static predicates. Inside of the predicate, the subtype name refers to the current instance of the subtype, which is an object, so a direct use of the subtype name cannot be recursive. But other subtypes naming the same type might: 24.a/3

```
   type Really_Ugly is private;                                          24.b/3
private
   subtype Ugly is Really_Ugly;
   type Really_Ugly is new Integer
      with Static_Predicate => Really_Ugly not in Ugly; -- Illegal!
```

{*AI05-0153-3*} An index subtype, discrete_range of an index_constraint or slice, or a discrete_subtype_definition of a constrained_array_definition, entry_declaration, or entry_index_specification shall not denote a subtype to which predicate specifications apply. 25/3

{*AI05-0153-3*} The prefix of an attribute_reference whose attribute_designator is First, Last, or Range shall not denote a scalar subtype to which predicate specifications apply. 26/3

Reason: {*AI05-0297-1*} This is to prevent confusion about whether the First value is the lowest value of the subtype (which does not depend on the predicate) or the lowest value of the subtype which meets the predicate. (For a dynamic predicate, determining this latter value is expensive as it would usually require a loop.) For a static subtype that has a static predicate, the First_Valid and Last_Valid attributes (see 3.5.5) can be used instead. 26.a/3

{*AI05-0153-3*} {*AI05-0262-1*} {*AI05-0287-1*} The discrete_subtype_definition of a loop_parameter_specification shall not denote a nonstatic subtype to which predicate specifications apply or any subtype to which Dynamic_Predicate specifications apply. 27/3

{*AI05-0153-3*} {*AI05-0262-1*} The discrete_choice of a named_array_aggregate shall not denote a nonstatic subtype to which predicate specifications apply. 28/3

Reason: {*AI05-0262-1*} This rule prevents noncontiguous dynamically bounded array aggregates, which could be expensive to check for. (Array aggregates have rules to prevent problems with static subtypes.) We define this rule here so that the runtime generic body check applies. 28.a/3

{*AI05-0262-1*} In addition to the places where Legality Rules normally apply (see 12.3), these rules apply also in the private part of an instance of a generic unit. 29/3

Dynamic Semantics

{*AI12-0071-1*} If any of the above Legality Rules is violated in an instance of a generic unit, Program_Error is raised at the point of the violation. 29.1/4

Discussion: This is the usual way around the contract model; this applies even in instance bodies. Note that errors in instance specifications will be detected at compile time by the "recheck" of the specification; only errors in the body should raise Program_Error. 29.a/4

29.2/4 {*AI12-0071-1*} To determine whether a value *satisfies the predicates* of a subtype *S*, the following tests are performed in the following order, until one of the tests fails, in which case the predicates are not satisfied and no further tests are performed, or all of the tests succeed, in which case the predicates are satisfied:

29.3/4 • the value is first tested to determine whether it satisfies any constraints or any null exclusion of *S*;

29.4/4 • then:

29.5/5 • {*AI12-0419-1*} if *S* is a first subtype, the value is tested to determine whether it satisfies the predicates of the parent and progenitor subtypes (if any) of *S* (in an arbitrary order), after a (view) conversion of the value to the corresponding parent or progenitor type;

29.b/4 **Ramification:** This rule has an effect for derived types (which have a parent subtype and may have progenitors) and for task and protected types (which may have progentitors). Other kinds of type declarations can have neither, and no test is required for other first subtypes.

29.6/4 • if *S* is defined by a subtype_indication, the value is tested to determine whether it satisfies the predicates of the subtype denoted by the subtype_mark of the subtype_indication;

29.7/4 • finally, if *S* is defined by a declaration to which one or more predicate specifications apply, the predicates are evaluated (in an arbitrary order) to test that all of them yield True for the given value.

29.b.1/4 **Discussion:** It is important to stop on the first of the above steps that fails, as later steps might presume that the earlier steps had succeeded.

30/3 {*AI05-0153-3*} {*AI05-0290-1*} If predicate checks are enabled for a given subtype, then:

31/5 {*AI12-0054-2*} {*AI12-0071-1*} {*AI12-0301-1*} {*AI12-0333-1*} {*AI12-0432-1*} [On a subtype conversion, a check is performed that the operand satisfies the predicates of the target subtype, except for certain view conversions (see 4.6).] In addition, after normal completion and leaving of a subprogram, for each **in out** or **out** parameter that is passed by reference, a check is performed that the value of the parameter satisfies the predicates of the subtype of the actual. For an object created by an object_declaration with no explicit initialization expression, or by an uninitialized allocator, if the types of any parts have specified Default_Value or Default_Component_Value aspects, or any subcomponents have default_expressions, a check is performed that the value of the created object satisfies the predicates of the nominal subtype.

31.a.1/5 **Ramification:** {*AI12-0333-1*} Most parameter passing is covered by the subtype conversion rule: all inbound **in** and **in out** parameters are converted to the formal subtype, and the copy-back for by-copy **out** and **in out** parameters includes a conversion to the actual subtype. The remaining parameter-passing cases are covered by special rules: by-reference **out** and **in out** parameters by the rule given above, and we don't want any predicate checks on inbound **out** parameters, accomplished in part by a special rule in 4.6.

31.1/4 {*AI12-0054-2*} If any of the predicate checks fail, Assertion_Error is raised, unless the subtype whose directly-specified predicate aspect evaluated to False also has a directly-specified Predicate_Failure aspect. In that case, the specified Predicate_Failure expression is evaluated; if the evaluation of the Predicate_Failure expression propagates an exception occurrence, then this occurrence is propagated for the failure of the predicate check; otherwise, Assertion_Error is raised, with an associated message string defined by the value of the Predicate_Failure expression. In the absence of such a Predicate_Failure aspect, an implementation-defined message string is associated with the Assertion_Error exception.

31.a/3 **Ramification:** Predicates are not evaluated at the point of the (sub)type declaration.

31.b/3 **Implementation Note:** Static_Predicate checks can be removed even in the presence of potentially invalid values, just as constraint checks can be removed.

31.c/4 **Implementation defined:** The message string associated with the Assertion_Error exception raised by the failure of a predicate check if there is no applicable Predicate_Failure aspect.

Paragraphs 32 and 33 were moved above

This paragraph was deleted. 33.a/4

NOTE 1 {*AI05-0153-3*} A predicate specification does not cause a subtype to be considered constrained. 34/3

NOTE 2 {*AI05-0153-3*} A Static_Predicate, like a constraint, always remains True for all objects of the subtype, except in the case of uninitialized variables and other invalid values. A Dynamic_Predicate, on the other hand, is checked as specified above, but can become False at other times. For example, the predicate of a record subtype is not checked when a subcomponent is modified. 35/3

NOTE 3 {*AI12-0071-1*} No predicates apply to the base subtype of a scalar type; every value of a scalar type *T* is considered to satisfy the predicates of *T*'Base. 36/4

NOTE 4 {*AI12-0054-2*} Predicate_Failure expressions are never evaluated during the evaluation of a membership test (see 4.5.2) or Valid attribute (see 13.9.2). 37/4

NOTE 5 {*AI12-0054-2*} A Predicate_Failure expression can be a raise_expression (see 11.3). 38/4

Examples

{AI12-0429-1} Examples of predicates applied to scalar types: 39/5

```
{AI12-0054-2} subtype Basic_Letter is Character -- See A.3.2 for "basic letter".
   with Static_Predicate => Basic_Letter in 'A'..'Z' | 'a'..'z' | 'Æ' |
                            'æ' | 'Ð' | 'ð' | 'Þ' | 'þ' | 'ß';
```
40/4

```
{AI12-0054-2} subtype Even_Integer is Integer
   with Dynamic_Predicate => Even_Integer mod 2 = 0,
        Predicate_Failure => "Even_Integer must be a multiple of 2";
```
41/4

{AI12-0054-2} Text_IO (see A.10.1) could have used predicates to describe some common exceptional conditions as follows: 42/4

```
with Ada.IO_Exceptions;
package Ada.Text_IO is
```
43/4

```
   type File_Type is limited private;
```
44/4

```
   subtype Open_File_Type is File_Type
      with Dynamic_Predicate => Is_Open (Open_File_Type),
           Predicate_Failure => raise Status_Error with "File not open";
   subtype Input_File_Type is Open_File_Type
      with Dynamic_Predicate => Mode (Input_File_Type) = In_File,
           Predicate_Failure => raise Mode_Error
              with "Cannot read file: " & Name (Input_File_Type);
   subtype Output_File_Type is Open_File_Type
      with Dynamic_Predicate => Mode (Output_File_Type) /= In_File,
           Predicate_Failure => raise Mode_Error
              with "Cannot write file: " & Name (Output_File_Type);
```
45/4

```
   ...
```
46/4

```
   function Mode (File : in Open_File_Type) return File_Mode;
   function Name (File : in Open_File_Type) return String;
   function Form (File : in Open_File_Type) return String;
```
47/4

```
   ...
```
48/4

```
   procedure Get (File : in Input_File_Type; Item : out Character);
```
49/4

```
   procedure Put (File : in Output_File_Type; Item : in Character);
```
50/4

```
   ...
```
51/4

```
   -- Similarly for all of the other input and output subprograms.
```
52/4

Discussion: We didn't change the language-defined Text_IO this way for Ada 2022 as it would be incompatible in marginal cases: these subprogram specifications would not be subtype conformant with existing access-to-subprogram types, so Put_Line'Access (for instance) would become illegal in existing code. The gain would not be worth the disruption. 52.a/5

Extensions to Ada 2005

{*AI05-0153-3*} {*AI05-0262-1*} {*AI05-0276-1*} {*AI05-0290-1*} Predicate aspects are new in Ada 2012. 52.b/3

Inconsistencies With Ada 2012

52.c/5 {*AI12-0301-1*} **Correction:** Predicate checks are now performed on default-initialized objects with parts that have Default_Value or Default_Component_Value specified. This is consistent with the handling of constraint checks for such objects; it is thought that the omission was unintended. However, a program that declares such an object and depends on there not being a predicate check in original Ada 2012 will fail in Ada 2022. As these attributes were new in Ada 2012, their use is uncommon, so we believe that this inconsistency will be rare and more likely to catch a bug than create one.

Extensions to Ada 2012

52.d/4 {*AI12-0054-2*} **Corrigendum:** The Predicate_Failure aspect is new. We can consider this a correction as it is always possible for implementers to add implementation-defined aspects, so the same is true for language-defined aspects.

Wording Changes from Ada 2012

52.e/4 {*AI12-0071-1*} **Corrigendum:** Specified the order of evaluation of most predicates, by defining the new term "satisfies the predicates of the subtype". This is not inconsistent, as the order previously was unspecified, so any code depending on the order was incorrect. The change is necessary so that the Predicate_Failure aspect has consistent results in cases where multiple predicates and aspects apply; see the Ada.Text_IO example above for such a case.

52.f/4 {*AI12-0099-1*} **Corrigendum:** Revised wording to ensure all kinds of types are covered, including the anonymous task associated with a single_task_declaration, and generalized it.

52.g/4 {*AI12-0099-1*} **Corrigendum:** Revised wording to list the boolean operators that can be predicate-static, to eliminate confusion about whether **not** is included.

52.h/5 {*AI12-0333-1*} {*AI12-0432-1*} **Correction:** Predicate checks are no longer made for any inbound **out** parameters nor for the target of an assignment_statement when it is a view conversion. The rule change for this is found in 4.6, so the inconsistency is documented there.

3.3 Objects and Named Numbers

1 [Objects are created at run time and contain a value of a given type. An object can be created and initialized as part of elaborating a declaration, evaluating an allocator, aggregate, or function_call, or passing a parameter by copy. Prior to reclaiming the storage for an object, it is finalized if necessary (see 7.6.1).]

Static Semantics

2 All of the following are objects:

2.a/5 **Term entry: object** — entity that contains a value, and is either a constant or a variable
Note: An object is created by an object_declaration or by an allocator. A formal parameter is (a view of) an object. A subcomponent of an object is an object.

3 • the entity declared by an object_declaration;

4 • a formal parameter of a subprogram, entry, or generic subprogram;

5 • a generic formal object;

6 • a loop parameter;

6.1/5 • {*AI12-0061-1*} the index parameter of an iterated_component_association;

6.2/5 • {*AI12-0308-1*} the chunk parameter of a chunk_specification;

7 • a choice parameter of an exception_handler;

8 • an entry index of an entry_body;

9 • the result of dereferencing an access-to-object value (see 4.1);

10/3 • {*AI95-00416-01*} {*AI05-0015-1*} the return object of a function;

11 • the result of evaluating an aggregate;

- {*AI05-0003-1*} {*AI12-0226-1*} a value conversion or qualified_expression whose operand denotes an object; 11.1/5
- a component, slice, or view conversion of another object. 12

{*AI05-0054-2*} An object is either a *constant* object or a *variable* object. Similarly, a view of an object is either a *constant* or a *variable*. All views of a constant elementary object are constant. All views of a constant composite object are constant, except for parts that are of controlled or immutably limited types; variable views of those parts and their subcomponents may exist. In this sense, objects of controlled and immutably limited types are *inherently mutable*. A constant view of an object cannot be used to modify its value. The terms constant and variable by themselves refer to constant and variable views of objects. 13/3

Ramification: {*AI12-0422-1*} If some part of an object has a variable view, then the object as a whole has a variable view, and not all views of the object are constant. That's true even if only a subcomponent has a variable view. 13.a/5

{*AI12-0422-1*} A constant object is *known to have no variable views* if it does not have a part that is immutably limited, or of a controlled type, private type, or private extension. 13.1/5

Reason: {*AI12-0422-1*} This definition can be used in Legality Rules as it respects privacy. It is an assume-the-worst rule, as all private types and private extensions might have a controlled component. 13.b/5

The value of an object is *read* when the value of any part of the object is evaluated, or when the value of an enclosing object is evaluated. The value of a variable is *updated* when an assignment is performed to any part of the variable, or when an assignment is performed to an enclosing object. 14

Ramification: Reading and updating are intended to include read/write references of any kind, even if they are not associated with the evaluation of a particular construct. Consider, for example, the expression "X.**all**(F)", where X is an access-to-array object, and F is a function. The implementation is allowed to first evaluate "X.**all**" and then F. Finally, a read is performed to get the value of the F'th component of the array. Note that the array is not necessarily read as part of the evaluation of "X.**all**". This is important, because if F were to free X using Unchecked_Deallocation, we want the execution of the final read to be erroneous. 14.a

{*AI12-0392-1*} Whether a view of an object is constant or variable is determined by the definition of the view. The following (and no others) represent variables: 15/5

- {*AI12-0392-1*} an object declared by an object_declaration without the reserved word **constant**; 16/5

  To be honest: {*AI95-00385-01*} We mean the word **constant** as defined by the grammar for object_declaration, not some random word **constant**. Thus, 16.a/2

  ```
  X : access constant T;
  ```
  16.b/2

  is not a constant. 16.c/2
- {*AI12-0392-1*} a formal parameter of mode **in out** or **out**; 17/5
- {*AI12-0392-1*} a generic formal object of mode **in out**; 17.1/5
- {*AI12-0392-1*} a non-discriminant component of a variable; 18/5

  Ramification: {*AI12-0392-1*} This includes both selected_components and indexed_components. 18.a/5
- {*AI12-0392-1*} a slice of a variable; 18.1/5
- {*AI05-0262-1*} {*AI12-0392-1*} a loop parameter that is specified to be a variable for a generalized loop (see 5.5.2); 18.2/5
- {*AI05-0262-1*} {*AI12-0392-1*} a view conversion of a variable; 19/5
- {*AI12-0392-1*} a dereference of an access-to-variable value; 20/5
- {*AI05-0015-1*} {*AI12-0392-1*} the return object declared by an extended_return_statement without the reserved word **constant**; 20.1/5
- {*AI05-0015-1*} {*AI12-0392-1*} the current instance of a type other than a protected type[, if the current instance is an object and not a value (see 8.6)]; 21/5

21.a/5 **Reason:** {*AI12-0392-1*} We exclude current instances of protected types as they are protected units and the next bullet applies.

21.b/5 **Proof:** {*AI12-0392-1*} This list of bullets only applies to views of objects, so current instances that are not objects are not considered here.

21.1/5 - *This paragraph was deleted.*{*AI05-0003-1*} {*AI12-0392-1*}

21.2/5 - {*AI05-0120-1*} {*AI12-0392-1*} the current instance of a protected unit except within the body of a protected function of that protected unit, or within a function declared immediately within the body of the protected unit;

22/5 - {*AI12-0392-1*} an attribute_reference where the attribute is defined to denote a variable (for example, the Storage_Pool attribute – see 13.11).

22.a/5 **Ramification:** {*AI12-0392-1*} In particular, this implies that the following are not variables:

22.b/5 - {*AI12-0226-1*} {*AI12-0392-1*} the result of evaluating a function_call, an aggregate, a value conversion, a qualified_expression, a conditional_expression, a raise_expression, or a parenthesized expression;

22.c/5 - {*AI12-0125-3*} a target_name of an assignment_statement (see 5.2.1);

22.d/5 - {*AI12-0061-1*} the index parameter of an iterated_component_association;

22.e/5 - {*AI12-0392-1*} a choice parameter or entry index;

22.f/5 - {*AI12-0308-1*} a chunk parameter of a chunk_specification.

22.g/5 {*AI12-0392-1*} This list of constructs that yield constant views is not exhaustive.

23/5 {*AI05-0264-1*} {*AI12-0191-1*} {*AI12-0294-1*} At the place where a view of an object is defined, a *nominal subtype* is associated with the view. The *nominal type* of a view is the type of the nominal subtype of the view. The object's *actual subtype* (that is, its subtype) can be more restrictive than the nominal subtype of the view; it always is more restrictive if the nominal subtype is an *indefinite subtype*. A subtype is an indefinite subtype if it is an unconstrained array subtype, or if it has unknown discriminants or unconstrained discriminants without defaults (see 3.7); otherwise, the subtype is a *definite* subtype [(all elementary subtypes are definite subtypes)]. [A class-wide subtype is defined to have unknown discriminants, and is therefore an indefinite subtype. An indefinite subtype does not by itself provide enough information to create an object; an additional constraint or explicit initialization expression is necessary (see 3.3.1). A component cannot have an indefinite nominal subtype.]

23.a/5 **Term entry: nominal subtype** — subtype specified when a view of an object is defined

23.1/3 {*AI05-0008-1*} A view of a composite object is *known to be constrained* if:

23.2/5 - {*AI12-0401-1*} its nominal subtype is constrained and not an untagged partial view, and it is neither a value conversion nor a qualified_expression; or

23.3/3 - its nominal subtype is indefinite; or

23.4/3 - {*AI05-0008-1*} {*AI05-0093-1*} its type is immutably limited (see 7.5); or

23.5/3 - it is part of a stand-alone constant (including a generic formal object of mode **in**); or

23.6/3 - it is part of a formal parameter of mode **in**; or

23.7/3 - it is part of the object denoted by a function_call or aggregate; or

23.8/5 - {*AI12-0226-1*} {*AI12-0228-1*} it is a value conversion or qualified_expression where the operand denotes a view of a composite object that is known to be constrained; or

23.9/5 - {*AI12-0228-1*} it is part of a constant return object of an extended_return_statement; or

23.10/5 - {*AI05-0008-1*} {*AI05-0041-1*} {*AI12-0228-1*} it is a dereference of a pool-specific access type, and there is no ancestor of its type that has a constrained partial view.

23.b/3 **Discussion:** We do not include dereferences of general access types because they might denote stand-alone aliased unconstrained variables. That's true even for access-to-constant types (the denoted object does not have to be a constant).

{*AI05-0005-1*} {*AI05-0008-1*} {*AI12-0228-1*} We don't mention view conversions as there are no mutable tagged types (discriminant defaults are allowed only if the type is immutably limited), so all tagged view conversions are either of an indefinite type (if it has discriminants without defaults), an immutably limited type (if the discriminants do have defaults), or constrained (if there are no discriminants). This matches the first three bullets here, so all tagged view conversions are known to be constrained without needing to mention them explicitly. Untagged view conversions only can occur in parameter passing (as actuals to **in out** or **out** parameters), and "known to be constrained" is not used there. 23.c/5

{*AI12-0228-1*} We don't need to mention the current instance of a (sub)type, either. If a current instance of a type or subtype appears in an aspect specification, it represents a value, so whether or not it is known to be constrained is irrelevant (the term is only defined for composite objects). Otherwise, the current instance of a type can only be used in an immutably limited type, so all such instances are known to be constrained by the third bullet. 23.d/5

{*AI05-0008-1*} {*AI05-0041-1*} {*AI12-0228-1*} For the purposes of determining within a generic body whether an object is known to be constrained: 23.11/5

- if a subtype is a descendant of an untagged generic formal private or derived type, and the subtype is not an unconstrained array subtype, it is not considered indefinite and is considered to have a constrained partial view; 23.12/5
- if a subtype is a descendant of a formal access type, it is not considered pool-specific. 23.13/5

A *named number* provides a name for a numeric value known at compile time. It is declared by a number_declaration. 24

NOTE 1 A constant cannot be the target of an assignment operation, nor be passed as an **in out** or **out** parameter, between its initialization and finalization, if any. 25

NOTE 2 {*AI05-0054-2*} The value of a constant object cannot be changed after its initialization, except in some cases where the object has a controlled or immutably limited part (see 7.5, 7.6, and 13.9.1). 25.1/3

NOTE 3 {*AI05-0264-1*} The nominal and actual subtypes of an elementary object are always the same. For a discriminated or array object, if the nominal subtype is constrained, then so is the actual subtype. 26/3

Extensions to Ada 83

There are additional kinds of objects (choice parameters and entry indices of entry bodies). 26.a

The result of a function and of evaluating an aggregate are considered (constant) objects. This is necessary to explain the action of finalization on such things. Because a function_call is also syntactically a name (see 4.1), the result of a function_call can be renamed, thereby allowing repeated use of the result without calling the function again. 26.b

Wording Changes from Ada 83

{*AI05-0299-1*} This subclause now follows the subclauses on types and subtypes, to cut down on the number of forward references. 26.c/3

The term nominal subtype is new. It is used to distinguish what is known at compile time about an object's constraint, versus what its "true" run-time constraint is. 26.d

The terms definite and indefinite (which apply to subtypes) are new. They are used to aid in the description of generic formal type matching, and to specify when an explicit initial value is required in an object_declaration. 26.e

We have moved the syntax for object_declaration and number_declaration down into their respective subclauses, to keep the syntax close to the description of the associated semantics. 26.f

We talk about variables and constants here, since the discussion is not specific to object_declarations, and it seems better to have the list of the kinds of constants juxtaposed with the kinds of objects. 26.g

We no longer talk about indirect updating due to parameter passing. Parameter passing is handled in 6.2 and 6.4.1 in a way that there is no need to mention it here in the definition of read and update. Reading and updating now includes the case of evaluating or assigning to an enclosing object. 26.h

Wording Changes from Ada 95

{*AI95-00416-01*} Clarified that the return object is the object created by a function call. 26.i/2

Extensions to Ada 2005

{*AI05-0015-1*} Added wording to allow return objects to be declared as constants, and corrected the definition of return objects as objects. 26.j/3

Wording Changes from Ada 2005

26.k/3 {*AI05-0008-1*} {*AI05-0041-1*} {*AI05-0093-1*} **Correction:** Added a definition of *known to be constrained*, for use in other rules.

26.l/3 {*AI05-0054-2*} **Correction:** We now recognize the fact that not all declared constant objects are immutable; for those that a variable view can be constructed, they can be changed via that view.

26.m/3 {*AI05-0120-1*} **Correction:** Added the current instance of a protected object to the list of constant views; since the list claims to include all possibilities, it had better include that one.

26.n/3 {*AI05-0003-1*} The result of a qualified_expression is defined to be a constant view and is defined to be an object if the operand of the qualified_expression is an object. These definitions, combined with some grammar changes, allow qualified_expressions to be used in more places. See 4.1 for details.

Incompatibilities With Ada 2012

26.o/5 {*AI12-0401-1*} **Correction:** Corrected the definition of "known to be constrained" so that the status of the operand of value conversions and qualified_expressions is always used to determine whether the property exists. As the rules are **or**ed together, a value conversion or qualified_expression with a constrained nominal subtype would have always met the requirements in Ada 2012, regardless of the operand. This change will mean that some conversions or qualifications (mostly of variables) will no longer be considered "known to be constrained" and therefore 'Access and renaming of such prefixes will now be illegal. This is necessary to meet the design goal that subsequent execution cannot cause a renaming or 'Access to cause erroneous execution.

Extensions to Ada 2012

26.p/5 {*AI12-0228-1*} **Correction:** A qualified_expression of an object that is known to be constrained is now also known to be constrained. This allows qualification to be used to disambiguate a function call used as a prefix in a renaming_declaration without making the renaming_declaration illegal.

26.q/5 {*AI12-0226-1*} A value conversion of an object is an object; this makes value conversions consistent with qualified expressions.

Wording Changes from Ada 2012

26.r/5 {*AI12-0392-1*} **Correction:** Changed from a list of constants to a list of variables. This makes the default to be a constant, which is the more common case, and eliminates issues caused by omissions from the list (such as parenthesized expressions).

26.s/5 {*AI12-0422-1*} Added the term "known to have no variable views" in order to have a definition that can be used in Legality Rules without breaking privacy.

3.3.1 Object Declarations

1/3 {*AI05-0262-1*} An object_declaration declares a *stand-alone* object with a given nominal subtype and, optionally, an explicit initial value given by an initialization expression. For an array, access, task, or protected object, the object_declaration may include the definition of the (anonymous) type of the object.

Syntax

2/3 {*AI95-00385-01*} {*AI95-00406-01*} {*AI05-0183-1*} object_declaration ::=
defining_identifier_list : [**aliased**] [**constant**] subtype_indication [:= expression]
[aspect_specification];
| defining_identifier_list : [**aliased**] [**constant**] access_definition [:= expression]
[aspect_specification];
| defining_identifier_list : [**aliased**] [**constant**] array_type_definition [:= expression]
[aspect_specification];
| single_task_declaration
| single_protected_declaration

3 defining_identifier_list ::=
defining_identifier {, defining_identifier}

Name Resolution Rules

For an object_declaration with an expression following the compound delimiter :=, the type expected for the expression is that of the object. This expression is called the *initialization expression*. 4

Legality Rules

{*AI95-00287-01*} An object_declaration without the reserved word **constant** declares a variable object. If it has a subtype_indication or an array_type_definition that defines an indefinite subtype, then there shall be an initialization expression. 5/2

Static Semantics

{*AI05-0264-1*} {*AI05-0299-1*} {*AI12-0449-1*} An object_declaration with the reserved word **constant** declares a constant object. If it has an initialization expression, then it is called a *full constant declaration*. Otherwise, it is called a *deferred constant declaration*. The rules for deferred constant declarations are given in 7.4. The rules for full constant declarations are given in this subclause. 6/5

Any declaration that includes a defining_identifier_list with more than one defining_identifier is equivalent to a series of declarations each containing one defining_identifier from the list, with the rest of the text of the declaration copied for each declaration in the series, in the same order as the list. The remainder of this Reference Manual relies on this equivalence; explanations are given for declarations with a single defining_identifier. 7

{*AI95-00385-01*} The subtype_indication, access_definition, or full type definition of an object_declaration defines the nominal subtype of the object. The object_declaration declares an object of the type of the nominal subtype. 8/2

Discussion: {*AI95-00385-01*} The phrase "full type definition" here includes the case of an anonymous array, access, task, or protected type. 8.a/2

{*AI95-00373-01*} {*AI12-0192-1*} A component of an object is said to *require late initialization* if: 8.1/5

- {*AI12-0192-1*} {*AI12-0404-1*} it has an access discriminant value constrained by a per-object expression; or 8.2/5
- {*AI12-0192-1*} it has an initialization expression that includes a name denoting an access discriminant; or 8.3/5
- {*AI12-0192-1*} it has an initialization expression that includes a reference to the current instance of the type either by name or implicitly as the target object of a call. 8.4/5

Reason: Such components can depend on the values of other components of the object. We want to initialize them as late and as reproducibly as possible. 8.b/2

Dynamic Semantics

{*AI95-00363-01*} If a composite object declared by an object_declaration has an unconstrained nominal subtype, then if this subtype is indefinite or the object is constant the actual subtype of this object is constrained. The constraint is determined by the bounds or discriminants (if any) of its initial value; the object is said to be *constrained by its initial value*. When not constrained by its initial value, the actual and nominal subtypes of the object are the same. If its actual subtype is constrained, the object is called a *constrained object*. 9/2

For an object_declaration without an initialization expression, any initial values for the object or its subcomponents are determined by the *implicit initial values* defined for its nominal subtype, as follows: 10

- The implicit initial value for an access subtype is the null value of the access type. 11

11.1/5 • {*AI05-0228-1*} {*AI12-0439-1*} The implicit initial value for a scalar subtype that has the Default_Value aspect specified is the value of that aspect converted to the nominal subtype (which can raise Constraint_Error — see 4.6, “Type Conversions”);

11.a.1/3 **Ramification:** This is a Dynamic Semantics rule, so the visibility of the aspect_specification is not relevant — if the full type for a private type has the Default_Value aspect specified, partial views of the type also have this implicit initial value.

12 • The implicit initial (and only) value for each discriminant of a constrained discriminated subtype is defined by the subtype.

13/5 • {*AI05-0228-1*} {*AI12-0439-1*} For a (definite) composite subtype, the implicit initial value of each component with a default_expression is obtained by evaluation of this expression and conversion to the component's nominal subtype (which can raise Constraint_Error), unless the component is a discriminant of a constrained subtype (the previous case), or is in an excluded variant (see 3.8.1). For each component that does not have a default_expression, if the composite subtype has the Default_Component_Value aspect specified, the implicit initial value is the value of that aspect converted to the component's nominal subtype; otherwise, any implicit initial values are those determined by the component's nominal subtype.

14 • For a protected or task subtype, there is an implicit component (an entry queue) corresponding to each entry, with its implicit initial value being an empty queue.

14.a **Implementation Note:** The implementation may add implicit components for its own use, which might have implicit initial values. For a task subtype, such components might represent the state of the associated thread of control. For a type with dynamic-sized components, such implicit components might be used to hold the offset to some explicit component.

15 The elaboration of an object_declaration proceeds in the following sequence of steps:

16/2 1. {*AI95-00385-01*} The subtype_indication, access_definition, array_type_definition, single_-task_declaration, or single_protected_declaration is first elaborated. This creates the nominal subtype (and the anonymous type in the last four cases).

17/5 2. {*AI12-0439-1*} If the object_declaration includes an initialization expression, the (explicit) initial value is obtained by evaluating the expression and converting it to the nominal subtype (which can raise Constraint_Error — see 4.6).

18/2 3. {*8652/0002*} {*AI95-00171-01*} {*AI95-00373-01*} The object is created, and, if there is not an initialization expression, the object is *initialized by default*. When an object is initialized by default, any per-object constraints (see 3.8) are elaborated and any implicit initial values for the object or for its subcomponents are obtained as determined by the nominal subtype. Any initial values (whether explicit or implicit) are assigned to the object or to the corresponding subcomponents. As described in 5.2 and 7.6, Initialize and Adjust procedures can be called.

18.a **Discussion:** For a per-object constraint that contains some per-object expressions and some non-per-object expressions, the values used for the constraint consist of the values of the non-per-object expressions evaluated at the point of the type_declaration, and the values of the per-object expressions evaluated at the point of the creation of the object.

18.b The elaboration of per-object constraints was presumably performed as part of the dependent compatibility check in Ada 83. If the object is of a limited type with an access discriminant, the access_definition is elaborated at this time (see 3.7).

18.c **Reason:** The reason we say that evaluating an explicit initialization expression happens before creating the object is that in some cases it is impossible to know the size of the object being created until its initial value is known, as in “X: String := Func_Call(...);”. The implementation can create the object early in the common case where the size can be known early, since this optimization is semantically neutral.

19/2 *This paragraph was deleted.*{*AI95-00373-01*}

19.a **Ramification:** Since the initial values have already been converted to the appropriate nominal subtype, the only Constraint_Errors that might occur as part of these assignments are for values outside their base range that are used to initialize unconstrained numeric subcomponents. See 3.5.

{*AI95-00373-01*} For the third step above, evaluations and assignments are performed in an arbitrary order subject to the following restrictions: 20/2

- {*AI95-00373-01*} Assignment to any part of the object is preceded by the evaluation of the value that is to be assigned. 20.1/2

  Reason: Duh. But we ought to say it. Note that, like any rule in the Reference Manual, it doesn't prevent an "as-if" optimization; as long as the semantics as observed from the program are correct, the compiler can generate any code it wants. 20.a.1/2

- {*AI95-00373-01*} The evaluation of a default_expression that includes the name of a discriminant is preceded by the assignment to that discriminant. 20.2/2

  Reason: Duh again. But we have to say this, too. It's odd that Ada 95 only required the default expressions to be evaluated before the discriminant is used; it says nothing about discriminant values that come from subtype_indications. 20.a.2/2

- {*AI95-00373-01*} The evaluation of the default_expression for any component that depends on a discriminant is preceded by the assignment to that discriminant. 20.3/2

  Reason: For example: 20.a

```
type R(D : Integer := F) is                                    20.b
    record
        S : String(1..D) := (others => G);
    end record;

X : R;                                                         20.c
```

  For the elaboration of the declaration of X, it is important that F be evaluated before the aggregate. 20.d

- {*AI95-00373-01*} {*AI05-0092-1*} The assignments to any components, including implicit components, not requiring late initialization precede the initial value evaluations for any components requiring late initialization; if two components both require late initialization, then assignments to parts of the component occurring earlier in the order of the component declarations precede the initial value evaluations of the component occurring later. 20.4/3

  Reason: Components that require late initialization can refer to the entire object during their initialization. We want them to be initialized as late as possible to reduce the chance that their initialization depends on uninitialized components. For instance: 20.e/2

```
type T (D : Natural) is                                        20.f/2
  limited record
    C1 : T1 (T'Access);
    C2 : Natural := F (D);
    C3 : String (1 .. D) := (others => ' ');
  end record;
```

  Component C1 requires late initialization. The initialization could depend on the values of any component of T, including D, C2, or C3. Therefore, we want to it to be initialized last. Note that C2 and C3 do not require late initialization; they only have to be initialized after D. 20.g/2

  It is possible for there to be more than one component that requires late initialization. In this case, the language can't prevent problems, because all of the components can't be the last one initialized. In this case, we specify the order of initialization for components requiring late initialization; by doing so, programmers can arrange their code to avoid accessing uninitialized components, and such arrangements are portable. Note that if the program accesses an uninitialized component, 13.9.1 defines the execution to be erroneous. 20.h/2

{*AI05-0228-1*} {*AI12-0439-1*} [There is no implicit initial value defined for a scalar subtype unless the Default_Value aspect has been specified for the type.] In the absence of an explicit initialization or the specification of the Default_Value aspect, a newly created scalar object can have a value that does not belong to its subtype (see 13.9.1 and H.1). 21/5

To be honest: It could even be represented by a bit pattern that doesn't actually represent any value of the type at all, such as an invalid internal code for an enumeration type, or a NaN for a floating point type. It is a generally a bounded error to reference scalar objects with such "invalid representations", as explained in 13.9.1, "Data Validity". 21.a

Ramification: There is no requirement that two objects of the same scalar subtype have the same implicit initial "value" (or representation). It might even be the case that two elaborations of the same object_declaration produce two different initial values. However, any particular uninitialized object is default-initialized to a single value (or invalid 21.b

representation). Thus, multiple reads of such an uninitialized object will produce the same value each time (if the implementation chooses not to detect the error).

22 NOTE 1 Implicit initial values are not defined for an indefinite subtype, because if an object's nominal subtype is indefinite, an explicit initial value is required.

23/5 NOTE 2 {*AI05-0092-1*} {*AI05-0255-1*} {*AI12-0061-1*} {*AI12-0308-1*} As indicated above, a stand-alone object is an object declared by an object_declaration. Similar definitions apply to "stand-alone constant" and "stand-alone variable". A subcomponent of an object is not a stand-alone object, nor is an object that is created by an allocator. An object declared by a loop_parameter_specification, iterator_specification, iterated_component_association, chunk_specification, parameter_specification, entry_index_specification, choice_parameter_specification, extended_return_statement, or a formal_object_declaration of mode **in out** is not considered a stand-alone object.

24 NOTE 3 The type of a stand-alone object cannot be abstract (see 3.9.3).

Examples

25 *Example of a multiple object declaration:*

26
```
-- the multiple object declaration
```

27/2
```
{AI95-00433-01} John, Paul : not null Person_Name := new Person(Sex => M); -- see 3.10.1
```

28
```
-- is equivalent to the two single object declarations in the order given
```

29/2
```
{AI95-00433-01} John : not null Person_Name := new Person(Sex => M);
Paul : not null Person_Name := new Person(Sex => M);
```

30 *Examples of variable declarations:*

31/5
```
{AI95-00433-01} {AI12-0430-1} Count, Sum  : Integer;
Size         : Integer range 0 .. 10_000 := 0;
Sorted       : Boolean := False;
Color_Table  : array(1 .. Max) of Color;
Option       : Bit_Vector(1 .. 10) := (others => True); -- see 3.6
Hello        : aliased String := "Hi, world.";
θ, φ         : Float range -π .. +π;
```

32 *Examples of constant declarations:*

33/5
```
{AI95-00433-01} {AI12-0425-1} Limit     : constant Integer := 10_000;
Low_Limit : constant Integer := Limit/10;
Tolerance : constant Real := Dispersion(1.15);
A_String  : constant String := "A";
Hello_Msg : constant access String := Hello'Access; -- see 3.10.2
```

Extensions to Ada 83

33.a The syntax rule for object_declaration is modified to allow the **aliased** reserved word.

33.b A variable declared by an object_declaration can be constrained by its initial value; that is, a variable of a nominally unconstrained array subtype, or discriminated type without defaults, can be declared so long as it has an explicit initial value. In Ada 83, this was permitted for constants, and for variables created by allocators, but not for variables declared by object_declarations. This is particularly important for tagged class-wide types, since there is no way to constrain them explicitly, and so an initial value is the only way to provide a constraint. It is also important for generic formal private types with unknown discriminants.

33.c We now allow an unconstrained_array_definition in an object_declaration. This allows an object of an anonymous array type to have its bounds determined by its initial value. This is for uniformity: If one can write "X: **constant array**(Integer **range** 1..10) **of** Integer := ...;" then it makes sense to also allow "X: **constant array**(Integer **range** <>) **of** Integer := ...;". (Note that if anonymous array types are ever sensible, a common situation is for a table implemented as an array. Tables are often constant, and for constants, there's usually no point in forcing the user to count the number of elements in the value.)

Wording Changes from Ada 83

33.d We have moved the syntax for object_declarations into this subclause.

33.e Deferred constants no longer have a separate syntax rule, but rather are incorporated in object_declaration as constants declared without an initialization expression.

Inconsistencies With Ada 95

{*AI95-00363-01*} Unconstrained aliased objects of types with discriminants with defaults are no longer constrained by their initial values. This means that a program that raised Constraint_Error from an attempt to change the discriminants will no longer do so. The change only affects programs that depended on the raising of Constraint_Error in this case, so the inconsistency is unlikely to occur outside of the ACATS. This change may however cause compilers to implement these objects differently, possibly taking additional memory or time. This is unlikely to be worse than the differences caused by any major compiler upgrade. 33.f/2

Extensions to Ada 95

{*AI95-00287-01*} A constant may have a limited type; the initialization expression has to be built-in-place (see 7.5). 33.g/2

{*AI95-00385-01*} {*AI95-00406-01*} A stand-alone object may have an anonymous access type. 33.h/2

Wording Changes from Ada 95

{*8652/0002*} {*AI95-00171-01*} **Corrigendum:** Corrected wording to say that per-object constraints are elaborated (not evaluated). 33.i/2

{*AI95-00373-01*} The rules for evaluating default initialization have been tightened. In particular, components whose default initialization can refer to the rest of the object are required to be initialized last. 33.j/2

{*AI95-00433-01*} Added examples of various new constructs. 33.k/2

Extensions to Ada 2005

{*AI05-0183-1*} An optional aspect_specification can be used in an object_declaration. This is described in 13.1.1. 33.l/3

Wording Changes from Ada 2005

{*AI05-0228-1*} Implicit initial values can now be given for scalar types and for scalar array components, using the Default_Value (see 3.5) and Default_Component_Value (see 3.6) aspects; the extension is documented there. 33.m/3

Inconsistencies With Ada 2012

{*AI12-0192-1*} **Correction:** Components of a protected type require late initialization if their initialization includes (implicitly) the current instance of the type. This means that the components could end up being initialized in a different order. In most cases, this will have no visible effect, or will even fix bugs. Most code for which this is an issue depends on the (unspecified) order of initialization, so it is at risk of failing with a new compiler version regardless of Ada rule changes. However, there do exist very unlikely cases where legal, portable Ada 2012 code would become erroneous. (See the discussion section of AI12-0192-1 for an example.) These are so unlikely that it is expected that they only exist in the minds of Ada lawyers. 33.n/5

3.3.2 Number Declarations

A number_declaration declares a named number. 1

Discussion: {*AI05-0299-1*} {*AI12-0449-1*} If a value or other property of a construct is required to be *static* that means it is required to be determined prior to execution. A *static* expression is an expression whose value is computed at compile time and is usable in contexts where the actual value might affect the legality of the construct. This is fully defined in 4.9. 1.a/5

Syntax

number_declaration ::= 2
    defining_identifier_list : **constant** := *static_*expression;

Name Resolution Rules

The *static_*expression given for a number_declaration is expected to be of any numeric type. 3

{*AI12-0394-1*} A name that denotes a number_declaration is interpreted as a value of a universal type, unless the expected type for the name is a non-numeric type with an Integer_Literal or Real_Literal aspect, in which case it is interpreted to be of its expected type. 3.1/5

3.a/5 **To be honest:** This is only a Name Resolution Rule; all named numbers are values of a universal type (see the Static Semantics below). We need this rule so that named numbers can match types with user-defined literals; we need the other rules so the value of the named number is well-defined in all cases.

Legality Rules

4/5 {*AI05-0299-1*} {*AI12-0449-1*} The *static_*expression given for a number declaration shall be a static expression, as defined by 4.9.

Static Semantics

5 The named number denotes a value of type *universal_integer* if the type of the *static_*expression is an integer type. The named number denotes a value of type *universal_real* if the type of the *static_*expression is a real type.

6 The value denoted by the named number is the value of the *static_*expression, converted to the corresponding universal type.

Dynamic Semantics

7 The elaboration of a number_declaration has no effect.

7.a **Proof:** Since the *static_*expression was evaluated at compile time.

Examples

8 *Examples of number declarations:*

9
```
Two_Pi        : constant := 2.0*Ada.Numerics.Pi;
                                              -- a real number (see A.5)
```

10/2
```
{AI95-00433-01} Max           : constant := 500;          -- an integer number
Max_Line_Size : constant := Max/6;        -- the integer 83
Power_16      : constant := 2**16;        -- the integer 65_536
One, Un, Eins : constant := 1;            -- three different names for 1
```

Extensions to Ada 83

10.a We now allow a static expression of any numeric type to initialize a named number. For integer types, it was possible in Ada 83 to use 'Pos to define a named number, but there was no way to use a static expression of some nonuniversal real type to define a named number. This change is upward compatible because of the preference rule for the operators of the root numeric types.

Wording Changes from Ada 83

10.b We have moved the syntax rule into this subclause.

10.c AI83-00263 describes the elaboration of a number declaration in words similar to that of an object_declaration. However, since there is no expression to be evaluated and no object to be created, it seems simpler to say that the elaboration has no effect.

Extensions to Ada 2012

10.d/5 {*AI12-0394-1*} Named numbers now can be used with (non-numeric) types that define user-defined literals (see 4.2.1).

3.4 Derived Types and Classes

1/2 {*AI95-00401-01*} {*AI95-00419-01*} A derived_type_definition defines a *derived type* (and its first subtype) whose characteristics are *derived* from those of a parent type, and possibly from progenitor types.

1.a/5 **Term entry: derived type** — type defined in terms of a parent type and zero or more progenitor types given in a derived type definition
Note 1: A derived type inherits properties such as components and primitive operations from its parent and progenitors.
Note 2: A type together with the types derived from it (directly or indirectly) form a derivation class.

{*AI95-00442-01*} A *class of types* is a set of types that is closed under derivation; that is, if the parent or a progenitor type of a derived type belongs to a class, then so does the derived type. By saying that a particular group of types forms a class, we are saying that all derivatives of a type in the set inherit the characteristics that define that set. The more general term *category of types* is used for a set of types whose defining characteristics are not necessarily inherited by derivatives; for example, limited, abstract, and interface are all categories of types, but not classes of types. 1.1/2

Ramification: A class of types is also a category of types. 1.b/2

Syntax

{*AI95-00251-01*} {*AI95-00419-01*} derived_type_definition ::= 2/2
[**abstract**] [**limited**] **new** *parent*_subtype_indication [[**and** interface_list] record_extension_part]

Legality Rules

{*AI95-00251-01*} {*AI95-00401-01*} {*AI95-00419-01*} The *parent*_subtype_indication defines the *parent subtype*; its type is the *parent type*. The interface_list defines the progenitor types (see 3.9.4). A derived type has one parent type and zero or more progenitor types. 3/2

Term entry: parent of a derived type — first ancestor type given in the definition of the derived type 3.a/5
Note: The parent can be almost any kind of type, including an interface type.

A type shall be completely defined (see 3.11.1) prior to being specified as the parent type in a derived_type_definition — [the full_type_declarations for the parent type and any of its subcomponents have to precede the derived_type_definition.] 4

Discussion: This restriction does not apply to the ancestor type of a private extension — see 7.3; such a type need not be completely defined prior to the private_extension_declaration. However, the restriction does apply to record extensions, so the ancestor type will have to be completely defined prior to the full_type_declaration corresponding to the private_extension_declaration. 4.a

Reason: We originally hoped we could relax this restriction. However, we found it too complex to specify the rules for a type derived from an incompletely defined limited type that subsequently became nonlimited. 4.b

{*AI95-00401-01*} If there is a record_extension_part, the derived type is called a *record extension* of the parent type. A record_extension_part shall be provided if and only if the parent type is a tagged type. [An interface_list shall be provided only if the parent type is a tagged type.] 5/2

Proof: {*AI95-00401-01*} The syntax only allows an interface_list to appear with a record_extension_part, and a record_extension_part can only be provided if the parent type is a tagged type. We give the last sentence anyway for completeness. 5.a.1/2

Implementation Note: We allow a record extension to inherit discriminants; an early version of Ada 9X did not. If the parent subtype is unconstrained, it can be implemented as though its discriminants were repeated in a new known_discriminant_part and then used to constrain the old ones one-for-one. However, in an extension aggregate, the discriminants in this case do not appear in the component association list. 5.a

Ramification: {*AI95-00114-01*} This rule needs to be rechecked in the visible part of an instance of a generic unit because of the "only if" part of the rule. For example: 5.b/2

```
generic                                                                    5.c/2
   type T is private;
package P is
   type Der is new T;
end P;

package I is new P (Some_Tagged_Type); -- illegal                          5.d/2
```

{*AI95-00114-01*} The instantiation is illegal because a tagged type is being extended in the visible part without a record_extension_part. Note that this is legal in the private part or body of an instance, both to avoid a contract model violation, and because no code that can see that the type is actually tagged can also see the derived type declaration. 5.e/2

No recheck is needed for derived types with a record_extension_part, as that has to be derived from something that is known to be tagged (otherwise the template is illegal). 5.f/2

5.1/3 {*AI95-00419-01*} {*AI05-0096-1*} If the reserved word **limited** appears in a derived_type_definition, the parent type shall be a limited type. If the parent type is a tagged formal type, then in addition to the places where Legality Rules normally apply (see 12.3), this rule applies also in the private part of an instance of a generic unit.

5.g/2 **Reason:** We allow **limited** because we don't inherit limitedness from interfaces, so we must have a way to derive a limited type from interfaces. The word **limited** has to be legal when the parent *could be* an interface, and that includes generic formal abstract types. Since we have to allow it in this case, we might as well allow it everywhere as documentation, to make it explicit that the type is limited.

5.h/2 However, we do not want to allow **limited** when the parent is nonlimited: limitedness cannot change in a derivation tree.

5.i/3 If the parent type is an untagged limited formal type with an actual type that is nonlimited, we allow derivation as a limited type in the private part or body as no place could have visibility on the resulting type where it was known to be nonlimited (outside of the instance). (See the previous paragraph's annotations for an explanation of this.) However, if the parent type is a tagged limited formal type with an actual type that is nonlimited, it would be possible to pass a value of the limited type extension to a class-wide type of the parent, which would be nonlimited. That's too weird to allow (even though all of the extension components would have to be nonlimited because the rules of 3.9.1 are rechecked), so we have a special rule to prevent that in the private part (type extension from a formal type is illegal in a generic package body).

Static Semantics

6 The first subtype of the derived type is unconstrained if a known_discriminant_part is provided in the declaration of the derived type, or if the parent subtype is unconstrained. Otherwise, the constraint of the first subtype *corresponds* to that of the parent subtype in the following sense: it is the same as that of the parent subtype except that for a range constraint (implicit or explicit), the value of each bound of its range is replaced by the corresponding value of the derived type.

6.a **Discussion:** A digits_constraint in a subtype_indication for a decimal fixed point subtype always imposes a range constraint, implicitly if there is no explicit one given. See 3.5.9, “Fixed Point Types”.

6.1/2 {*AI95-00231-01*} The first subtype of the derived type excludes null (see 3.10) if and only if the parent subtype excludes null.

7/3 {*AI05-0110-1*} The *characteristics* and implicitly declared primitive subprograms of the derived type are defined as follows:

7.a/3 **Ramification:** {*AI05-0110-1*} The characteristics of a type do not include its primitive subprograms (primitive subprograms include predefined operators). The rules governing availability/visibility and inheritance of characteristics are separate from those for primitive subprograms.

8/2 - {*AI95-00251-01*} {*AI95-00401-01*} {*AI95-00442-01*} [If the parent type or a progenitor type belongs to a class of types, then the derived type also belongs to that class.] The following sets of types, as well as any higher-level sets composed from them, are classes in this sense[, and hence the characteristics defining these classes are inherited by derived types from their parent or progenitor types]: signed integer, modular integer, ordinary fixed, decimal fixed, floating point, enumeration, boolean, character, access-to-constant, general access-to-variable, pool-specific access-to-variable, access-to-subprogram, array, string, non-array composite, nonlimited, untagged record, tagged, task, protected, and synchronized tagged.

8.a **Discussion:** This is inherent in our notion of a “class” of types. It is not mentioned in the initial definition of “class” since at that point type derivation has not been defined. In any case, this rule ensures that every class of types is closed under derivation.

9 - If the parent type is an elementary type or an array type, then the set of possible values of the derived type is a copy of the set of possible values of the parent type. For a scalar type, the base range of the derived type is the same as that of the parent type.

9.a **Discussion:** The base range of a type defined by an integer_type_definition or a real_type_definition is determined by the _definition, and is not necessarily the same as that of the corresponding root numeric type from which the newly defined type is implicitly derived. Treating numerics types as implicitly derived from one of the two root numeric

types is simply to link them into a type hierarchy; such an implicit derivation does not follow all the rules given here for an explicit derived_type_definition.

- If the parent type is a composite type other than an array type, then the components, protected subprograms, and entries that are declared for the derived type are as follows: 10
  - The discriminants specified by a new known_discriminant_part, if there is one; otherwise, each discriminant of the parent type (implicitly declared in the same order with the same specifications) — in the latter case, the discriminants are said to be *inherited*, or if unknown in the parent, are also unknown in the derived type; 11
  - Each nondiscriminant component, entry, and protected subprogram of the parent type, implicitly declared in the same order with the same declarations; these components, entries, and protected subprograms are said to be *inherited*; 12

    Ramification: The profiles of entries and protected subprograms do not change upon type derivation, although the type of the "implicit" parameter identified by the prefix of the name in a call does. 12.a

    To be honest: Any name in the parent type_declaration that denotes the current instance of the type is replaced with a name denoting the current instance of the derived type, converted to the parent type. 12.b
  - Each component declared in a record_extension_part, if any. 13

  Declarations of components, protected subprograms, and entries, whether implicit or explicit, occur immediately within the declarative region of the type, in the order indicated above, following the parent subtype_indication. 14

    Discussion: The order of declarations within the region matters for record_aggregates and extension_aggregates. 14.a

    Ramification: In most cases, these things are implicitly declared *immediately* following the parent subtype_indication. However, 7.3.1, "Private Operations" defines some cases in which they are implicitly declared later, and some cases in which the are not declared at all. 14.b

    Discussion: The place of the implicit declarations of inherited components matters for visibility — they are not visible in the known_discriminant_part nor in the parent subtype_indication, but are usually visible within the record_extension_part, if any (although there are restrictions on their use). Note that a discriminant specified in a new known_discriminant_part is not considered "inherited" even if it has the same name and subtype as a discriminant of the parent type. 14.c
- *This paragraph was deleted.*{*AI95-00419-01*} 15/2
- [For each predefined operator of the parent type, there is a corresponding predefined operator of the derived type.] 16

    Proof: This is a ramification of the fact that each class that includes the parent type also includes the derived type, and the fact that the set of predefined operators that is defined for a type, as described in 4.5, is determined by the classes to which it belongs. 16.a

    Reason: Predefined operators are handled separately because they follow a slightly different rule than user-defined primitive subprograms. In particular the systematic replacement described below does not apply fully to the relational operators for Boolean and the exponentiation operator for Integer. The relational operators for a type derived from Boolean still return Standard.Boolean. The exponentiation operator for a type derived from Integer still expects Standard.Integer for the right operand. In addition, predefined operators "reemerge" when a type is the actual type corresponding to a generic formal type, so they need to be well defined even if hidden by user-defined primitive subprograms. 16.b
- {*AI95-00401-01*} For each user-defined primitive subprogram (other than a user-defined equality operator — see below) of the parent type or of a progenitor type that already exists at the place of the derived_type_definition, there exists a corresponding *inherited* primitive subprogram of the derived type with the same defining name. Primitive user-defined equality operators of the parent type and any progenitor types are also inherited by the derived type, except when the derived type is a nonlimited record extension, and the inherited operator would have a profile that is type conformant with the profile of the corresponding predefined equality operator; in this case, the user-defined equality operator is not inherited, but is rather incorporated into the implementation of the predefined equality operator of the record extension (see 4.5.2). 17/2

17.a **Ramification:** We say "...already exists..." rather than "is visible" or "has been declared" because there are certain operations that are declared later, but still exist at the place of the derived_type_definition, and there are operations that are never declared, but still exist. These cases are explained in 7.3.1.

17.b Note that nonprivate extensions can appear only after the last primitive subprogram of the parent — the freezing rules ensure this.

17.c **Reason:** A special case is made for the equality operators on nonlimited record extensions because their predefined equality operators are already defined in terms of the primitive equality operator of their parent type (and of the tagged components of the extension part). Inheriting the parent's equality operator as is would be undesirable, because it would ignore any components of the extension part. On the other hand, if the parent type is limited, then any user-defined equality operator is inherited as is, since there is no predefined equality operator to take its place.

17.d/2 **Ramification:** {*AI95-00114-01*} Because user-defined equality operators are not inherited by nonlimited record extensions, the formal parameter names of = and /= revert to Left and Right, even if different formal parameter names were used in the user-defined equality operators of the parent type.

17.e/2 **Discussion:** {*AI95-00401-01*} This rule only describes what operations are inherited; the rules that describe what happens when there are conflicting inherited subprograms are found in 8.3.

18/3 {*AI95-00401-01*} {*AI05-0164-1*} {*AI05-0240-1*} The profile of an inherited subprogram (including an inherited enumeration literal) is obtained from the profile of the corresponding (user-defined) primitive subprogram of the parent or progenitor type, after systematic replacement of each subtype of its profile (see 6.1) that is of the parent or progenitor type, other than those subtypes found in the designated profile of an access_definition, with a *corresponding subtype* of the derived type. For a given subtype of the parent or progenitor type, the corresponding subtype of the derived type is defined as follows:

19 - If the declaration of the derived type has neither a known_discriminant_part nor a record_extension_part, then the corresponding subtype has a constraint that corresponds (as defined above for the first subtype of the derived type) to that of the given subtype.

20 - If the derived type is a record extension, then the corresponding subtype is the first subtype of the derived type.

21 - If the derived type has a new known_discriminant_part but is not a record extension, then the corresponding subtype is constrained to those values that when converted to the parent type belong to the given subtype (see 4.6).

21.a **Reason:** An inherited subprogram of an untagged type has an Intrinsic calling convention, which precludes the use of the Access attribute. We preclude 'Access because correctly performing all required constraint checks on an indirect call to such an inherited subprogram was felt to impose an undesirable implementation burden.

21.b/3 {*AI05-0164-1*} Note that the exception to substitution of the parent or progenitor type applies only in the profiles of anonymous access-to-subprogram types. The exception is necessary to avoid calling an access-to-subprogram with types and/or constraints different than expected by the actual routine.

22/2 {*AI95-00401-01*} The same formal parameters have default_expressions in the profile of the inherited subprogram. [Any type mismatch due to the systematic replacement of the parent or progenitor type by the derived type is handled as part of the normal type conversion associated with parameter passing — see 6.4.1.]

22.a/2 **Reason:** {*AI95-00401-01*} We don't introduce the type conversion explicitly here since conversions to record extensions or on access parameters are not generally legal. Furthermore, any type conversion would just be "undone" since the subprogram of the parent or progenitor is ultimately being called anyway. (Null procedures can be inherited from a progenitor without being overridden, so it is possible to call subprograms of an interface.)

23/2 {*AI95-00401-01*} If a primitive subprogram of the parent or progenitor type is visible at the place of the derived_type_definition, then the corresponding inherited subprogram is implicitly declared immediately after the derived_type_definition. Otherwise, the inherited subprogram is implicitly declared later or not at all, as explained in 7.3.1.

24 A derived type can also be defined by a private_extension_declaration (see 7.3) or a formal_derived_type_definition (see 12.5.1). Such a derived type is a partial view of the corresponding full or actual type.

All numeric types are derived types, in that they are implicitly derived from a corresponding root numeric type (see 3.5.4 and 3.5.6). 25

Dynamic Semantics

The elaboration of a derived_type_definition creates the derived type and its first subtype, and consists of the elaboration of the subtype_indication and the record_extension_part, if any. If the subtype_indication depends on a discriminant, then only those expressions that do not depend on a discriminant are evaluated. 26

Discussion: {*AI95-00251-01*} We don't mention the interface_list, because it does not need elaboration (see 3.9.4). This is consistent with the handling of discriminant_parts, which aren't elaborated either. 26.a/2

{*AI95-00391-01*} {*AI95-00401-01*} For the execution of a call on an inherited subprogram, a call on the corresponding primitive subprogram of the parent or progenitor type is performed; the normal conversion of each actual parameter to the subtype of the corresponding formal parameter (see 6.4.1) performs any necessary type conversion as well. If the result type of the inherited subprogram is the derived type, the result of calling the subprogram of the parent or progenitor is converted to the derived type, or in the case of a null extension, extended to the derived type using the equivalent of an extension_aggregate with the original result as the ancestor_part and **null record** as the record_component_association_list. 27/2

Discussion: {*AI95-00391-01*} If an inherited function returns the derived type, and the type is a nonnull record extension, then the inherited function shall be overridden, unless the type is abstract (in which case the function is abstract, and (unless overridden) cannot be called except via a dispatching call). See 3.9.3. 27.a/2

NOTE 1 Classes are closed under derivation — any class that contains a type also contains its derivatives. Operations available for a given class of types are available for the derived types in that class. 28

NOTE 2 Evaluating an inherited enumeration literal is equivalent to evaluating the corresponding enumeration literal of the parent type, and then converting the result to the derived type. This follows from their equivalence to parameterless functions. 29

NOTE 3 A generic subprogram is not a subprogram, and hence cannot be a primitive subprogram and cannot be inherited by a derived type. On the other hand, an instance of a generic subprogram can be a primitive subprogram, and hence can be inherited. 30

NOTE 4 If the parent type is an access type, then the parent and the derived type share the same storage pool; there is a **null** access value for the derived type and it is the implicit initial value for the type. See 3.10. 31

NOTE 5 If the parent type is a boolean type, the predefined relational operators of the derived type deliver a result of the predefined type Boolean (see 4.5.2). If the parent type is an integer type, the right operand of the predefined exponentiation operator is of the predefined type Integer (see 4.5.6). 32

NOTE 6 Any discriminants of the parent type are either all inherited, or completely replaced with a new set of discriminants. 33

NOTE 7 {*AI12-0442-1*} For an inherited subprogram, the subtype of a formal parameter of the derived type can be such that it has no value in common with the first subtype of the derived type. 34/5

Proof: This happens when the parent subtype is constrained to a range that does not overlap with the range of a subtype of the parent type that appears in the profile of some primitive subprogram of the parent type. For example: 34.a

```
type T1 is range 1..100;                                                    34.b
subtype S1 is T1 range 1..10;
procedure P(X : in S1);  -- P is a primitive subprogram
type T2 is new T1 range 11..20;
-- implicitly declared:
-- procedure P(X : in T2'Base range 1..10);
--     X cannot be in T2'First .. T2'Last
```

NOTE 8 If the reserved word **abstract** is given in the declaration of a type, the type is abstract (see 3.9.3). 35

NOTE 9 {*AI95-00251-01*} {*AI95-00401-01*} An interface type that has a progenitor type "is derived from" that type. A derived_type_definition, however, never defines an interface type. 35.1/2

NOTE 10 {*AI95-00345-01*} It is illegal for the parent type of a derived_type_definition to be a synchronized tagged type. 35.2/2

Proof: {*AI05-0299-1*} 3.9.1 prohibits record extensions whose parent type is a synchronized tagged type, and this subclause requires tagged types to have a record extension. Thus there are no legal derivations. Note that a 35.a/3

synchronized interface can be used as a progenitor in an interface_type_definition as well as in task and protected types, but we do not allow concrete extensions of any synchronized tagged type.

Examples

36 *Examples of derived type declarations:*

37
```
type Local_Coordinate is new Coordinate;  -- two different types
type Midweek is new Day range Tue .. Thu; -- see 3.5.1
type Counter is new Positive;             -- same range as Positive
```

38
```
type Special_Key is new Key_Manager.Key;  -- see 7.3.1
   -- the inherited subprograms have the following specifications:
   --      procedure Get_Key(K : out Special_Key);
   --      function "<"(X,Y : Special_Key) return Boolean;
```

Inconsistencies With Ada 83

38.a When deriving from a (nonprivate, nonderived) type in the same visible part in which it is defined, if a predefined operator had been overridden prior to the derivation, the derived type will inherit the user-defined operator rather than the predefined operator. The work-around (if the new behavior is not the desired behavior) is to move the definition of the derived type prior to the overriding of any predefined operators.

Incompatibilities With Ada 83

38.b When deriving from a (nonprivate, nonderived) type in the same visible part in which it is defined, a primitive subprogram of the parent type declared before the derived type will be inherited by the derived type. This can cause upward incompatibilities in cases like this:

38.c
```
      package P is
         type T is (A, B, C, D);
         function F( X : T := A ) return Integer;
         type NT is new T;
         -- inherits F as
         -- function F( X : NT := A ) return Integer;
         -- in Ada 95 only
         ...
      end P;
      ...
      use P;   -- Only one declaration of F from P is use-visible in
               -- Ada 83;  two declarations of F are use-visible in
               -- Ada 95.
   begin
      ...
      if F > 1 then ...  -- legal in Ada 83, ambiguous in Ada 95
```

Extensions to Ada 83

38.d The syntax for a derived_type_definition is amended to include an optional record_extension_part (see 3.9.1).

38.e A derived type may override the discriminants of the parent by giving a new discriminant_part.

38.f The parent type in a derived_type_definition may be a derived type defined in the same visible part.

38.g When deriving from a type in the same visible part in which it is defined, the primitive subprograms declared prior to the derivation are inherited as primitive subprograms of the derived type. See 3.2.3.

Wording Changes from Ada 83

38.h We now talk about the classes to which a type belongs, rather than a single class.

Extensions to Ada 95

38.i/2 {*AI05-0190-1*} {*AI95-00251-01*} {*AI95-00401-01*} A derived type may inherit from multiple (interface) progenitors, as well as the parent type — see 3.9.4, “Interface Types”.

38.j/2 {*AI95-00419-01*} A derived type may specify that it is a limited type. This is required for interface ancestors (from which limitedness is not inherited), but it is generally useful as documentation of limitedness.

Wording Changes from Ada 95

38.k/2 {*AI95-00391-01*} Defined the result of functions for null extensions (which we no longer require to be overridden - see 3.9.3).

{*AI95-00442-01*} Defined the term “category of types” and used it in wording elsewhere; also specified the language-defined categories that form classes of types (this was never normatively specified in Ada 95). 38.l/2

Incompatibilities With Ada 2005

{*AI05-0096-1*} **Correction:** Added a (re)check that limited type extensions never are derived from nonlimited types in generic private parts. This is disallowed as it would make it possible to pass a limited object to a nonlimited class-wide type, which could then be copied. This is only possible using Ada 2005 syntax, so examples in existing programs should be rare. 38.m/3

Wording Changes from Ada 2005

{*AI05-0110-1*} **Correction:** Added wording to clarify that the characteristics of derived types are formally defined here. (This is the only place in the Reference Manual that actually spells out what sorts of things are actually characteristics, which is rather important.) 38.n/3

{*AI05-0164-1*} **Correction:** Added wording to ensure that anonymous access-to-subprogram types don't get modified on derivation. 38.o/3

3.4.1 Derivation Classes

In addition to the various language-defined classes of types, types can be grouped into *derivation classes*. 1

Static Semantics

{*AI95-00251-01*} {*AI95-00401-01*} A derived type is *derived from* its parent type *directly*; it is derived *indirectly* from any type from which its parent type is derived. A derived type, interface type, type extension, task type, protected type, or formal derived type is also derived from every ancestor of each of its progenitor types, if any. The derivation class of types for a type *T* (also called the class *rooted* at *T*) is the set consisting of *T* (the *root type* of the class) and all types derived from *T* (directly or indirectly) plus any associated universal or class-wide types (defined below). 2/2

Discussion: Note that the definition of “derived from” is a recursive definition. We don't define a root type for all interesting language-defined classes, though presumably we could. 2.a

To be honest: By the class-wide type “associated” with a type *T*, we mean the type *T*'Class. Similarly, the universal type associated with *root_integer*, *root_real*, and *root_fixed* are *universal_integer*, *universal_real*, and *universal_fixed*, respectively. 2.b

{*AI95-00230-01*} {*AI12-0437-1*} Every type is one of a *specific* type, a *class-wide* type, or a *universal* type. A specific type is one defined by a type_declaration, a formal_type_declaration, or a full type definition embedded in another construct. Class-wide and universal types are implicitly defined, to act as representatives for an entire class of types, as follows: 3/5

To be honest: The root types *root_integer*, *root_real*, and *root_fixed* are also specific types. They are declared in the specification of package Standard. 3.a

Class-wide types 4

Class-wide types are defined for [(and belong to)] each derivation class rooted at a tagged type (see 3.9). Given a subtype S of a tagged type *T*, S'Class is the subtype_mark for a corresponding subtype of the tagged class-wide type *T*'Class. Such types are called “class-wide” because when a formal parameter is defined to be of a class-wide type *T*'Class, an actual parameter of any type in the derivation class rooted at *T* is acceptable (see 8.6).

The set of values for a class-wide type *T*'Class is the discriminated union of the set of values of each specific type in the derivation class rooted at *T* (the tag acts as the implicit discriminant — see 3.9). Class-wide types have no primitive subprograms of their own. However, as explained in 3.9.2, operands of a class-wide type *T*'Class can be used as part of a dispatching call on a primitive subprogram of the type *T*. The only components [(including discriminants)] of *T*'Class that are visible are those of *T*. If S is a first subtype, then S'Class is a first subtype. 5

5.a **Reason:** We want S'Class to be a first subtype when S is, so that an attribute_definition_clause like "**for** S'Class'Output **use** ...;" will be legal.

6/5 {*AI95-00230-01*} {*AI12-0445-1*} Universal types

Universal types are defined for [(and belong to)] the integer, real, fixed point, and access classes, and are referred to in this document as respectively, *universal_integer*, *universal_real*, *universal_fixed*, and *universal_access*. These are analogous to class-wide types for these language-defined elementary classes. As with class-wide types, if a formal parameter is of a universal type, then an actual parameter of any type in the corresponding class is acceptable. In addition, a value of a universal type (including an integer or real numeric_literal, or the literal **null**) is "universal" in that it is acceptable where some particular type in the class is expected (see 8.6).

7 The set of values of a universal type is the undiscriminated union of the set of values possible for any definable type in the associated class. Like class-wide types, universal types have no primitive subprograms of their own. However, their "universality" allows them to be used as operands with the primitive subprograms of any type in the corresponding class.

7.a **Discussion:** A class-wide type is only class-wide in one direction, from specific to class-wide, whereas a universal type is class-wide (universal) in both directions, from specific to universal and back.

7.b/2 {*AI95-00230-01*} We considered defining class-wide or perhaps universal types for all derivation classes, not just tagged classes and these four elementary classes. However, this was felt to overly weaken the strong-typing model in some situations. Tagged types preserve strong type distinctions thanks to the run-time tag. Class-wide or universal types for untagged types would weaken the compile-time type distinctions without providing a compensating run-time-checkable distinction.

7.c We considered defining standard names for the universal numeric types so they could be used in formal parameter specifications. However, this was felt to impose an undue implementation burden for some implementations.

7.d **To be honest:** Formally, the set of values of a universal type is actually a *copy* of the undiscriminated union of the values of the types in its class. This is because we want each value to have exactly one type, with explicit or implicit conversion needed to go between types. An alternative, consistent model would be to associate a class, rather than a particular type, with a value, even though any given expression would have a particular type. In that case, implicit type conversions would not generally need to change the value, although an associated subtype conversion might need to.

8 The integer and real numeric classes each have a specific root type in addition to their universal type, named respectively *root_integer* and *root_real*.

9/5 {*AI12-0325-1*} A class-wide or universal type is said to *cover* all of the types in its class. In addition, *universal_integer* covers a type that has a specified Integer_Literal aspect, while *universal_real* covers a type that has a specified Real_Literal aspect (see 4.2.1). A specific type covers only itself.

10/2 {*AI95-00230-01*} {*AI95-00251-01*} A specific type *T2* is defined to be a *descendant* of a type *T1* if *T2* is the same as *T1*, or if *T2* is derived (directly or indirectly) from *T1*. A class-wide type *T2*'Class is defined to be a descendant of type *T1* if *T2* is a descendant of *T1*. Similarly, the numeric universal types are defined to be descendants of the root types of their classes. If a type *T2* is a descendant of a type *T1*, then *T1* is called an *ancestor* of *T2*. An *ultimate ancestor* of a type is an ancestor of that type that is not itself a descendant of any other type. Every untagged type has a unique ultimate ancestor.

10.a **Ramification:** A specific type is a descendant of itself. Class-wide types are considered descendants of the corresponding specific type, and do not have any descendants of their own.

10.b A specific type is an ancestor of itself. The root of a derivation class is an ancestor of all types in the class, including any class-wide types in the class.

10.c **Discussion:** The terms root, parent, ancestor, and ultimate ancestor are all related. For example:

10.d/2 - {*AI95-00251-01*} Each type has at most one parent, and one or more ancestor types; each untagged type has exactly one ultimate ancestor. In Ada 83, the term "parent type" was sometimes used more generally to include any ancestor type (e.g. RM83-9.4(14)). In Ada 95, we restrict parent to mean the immediate ancestor.

10.e - A class of types has at most one root type; a derivation class has exactly one root type.

10.f - The root of a class is an ancestor of all of the types in the class (including itself).

- The type *root_integer* is the root of the integer class, and is the ultimate ancestor of all integer types. A similar statement applies to *root_real*. 10.g

Term entry: ancestor of a type — type itself or, in the case of a type derived from other types, its parent type or one of its progenitor types or one of their ancestors 10.h/5
Note: Ancestor and descendant are inverse relationships.

Term entry: descendant of a type — type itself or a type derived (directly or indirectly) from it 10.i/5
Note: Descendant and ancestor are inverse relationships.

An inherited component [(including an inherited discriminant)] of a derived type is inherited *from* a given ancestor of the type if the corresponding component was inherited by each derived type in the chain of derivations going back to the given ancestor. 11

NOTE Because operands of a universal type are acceptable to the predefined operators of any type in their class, ambiguity can result. For *universal_integer* and *universal_real*, this potential ambiguity is resolved by giving a preference (see 8.6) to the predefined operators of the corresponding root types (*root_integer* and *root_real*, respectively). Hence, in an apparently ambiguous expression like 12

1 + 4 < 7 13

where each of the literals is of type *universal_integer*, the predefined operators of *root_integer* will be preferred over those of other specific integer types, thereby resolving the ambiguity. 14

Ramification: Except for this preference, a root numeric type is essentially like any other specific type in the associated numeric class. In particular, the result of a predefined operator of a root numeric type is not "universal" (implicitly convertible) even if both operands were. 14.a

Wording Changes from Ada 95

{*AI95-00230-01*} Updated the wording to define the *universal_access* type. This was defined to make **null** for anonymous access types sensible. 14.b/2

{*AI95-00251-01*} {*AI95-00401-01*} The definitions of ancestors and descendants were updated to allow multiple ancestors (necessary to support interfaces). 14.c/2

Wording Changes from Ada 2012

{*AI12-0325-1*} Updated the wording to say that universal types cover the types with the appropriate user-defined literal. 14.d/5

3.5 Scalar Types

Scalar types comprise enumeration types, integer types, and real types. Enumeration types and integer types are called *discrete* types; each value of a discrete type has a *position number* which is an integer value. Integer types and real types are called *numeric* types. [All scalar types are ordered, that is, all relational operators are predefined for their values.] 1

Syntax

range_constraint ::= **range** range 2

range ::= range_attribute_reference 3
  | simple_expression .. simple_expression

Discussion: These need to be simple_expressions rather than more general expressions because ranges appear in membership tests and other contexts where expression .. expression would be ambiguous. 3.a

A *range* has a *lower bound* and an *upper bound* and specifies a subset of the values of some scalar type (the *type of the range*). A range with lower bound L and upper bound R is described by "L .. R". If R is less than L, then the range is a *null range*, and specifies an empty set of values. Otherwise, the range specifies the values of the type from the lower bound to the upper bound, inclusive. A value *belongs* to a range if it is of the type of the range, and is in the subset of values specified by the range. A value *satisfies* a range constraint if it belongs to the associated range. One range is *included* in another if all values that belong to the first range also belong to the second. 4

Name Resolution Rules

5 For a subtype_indication containing a range_constraint, either directly or as part of some other scalar_constraint, the type of the range shall resolve to that of the type determined by the subtype_mark of the subtype_indication. For a range of a given type, the simple_expressions of the range (likewise, the simple_expressions of the equivalent range for a range_attribute_reference) are expected to be of the type of the range.

5.a **Discussion:** In Ada 95, constraints only appear within subtype_indications; things that look like constraints that appear in type declarations are called something else like real_range_specifications.

5.b/5 {*AI05-0299-1*} {*AI12-0449-1*} We say "the expected type is ..." or "the type is expected to be ..." depending on which reads better. They are fundamentally equivalent, and both feed into the type resolution rules of 8.6.

5.c In some cases, it doesn't work to use expected types. For example, in the above rule, we say that the “type of the range shall resolve to ...” rather than “the expected type for the range is ...” We then use “expected type” for the bounds. If we used “expected” at both points, there would be an ambiguity, since one could apply the rules of 8.6 either on determining the type of the range, or on determining the types of the individual bounds. It is clearly important to allow one bound to be of a universal type, and the other of a specific type, so we need to use “expected type” for the bounds. Hence, we used “shall resolve to” for the type of the range as a whole. There are other situations where “expected type” is not quite right, and we use “shall resolve to” instead.

Static Semantics

6 The *base range* of a scalar type is the range of finite values of the type that can be represented in every unconstrained object of the type; it is also the range supported at a minimum for intermediate values during the evaluation of expressions involving predefined operators of the type.

6.a **Implementation Note:** Note that in some machine architectures intermediates in an expression (particularly if static), and register-resident variables might accommodate a wider range. The base range does not include the values of this wider range that are not assignable without overflow to memory-resident objects.

6.b **Ramification:** The base range of an enumeration type is the range of values of the enumeration type.

6.c **Reason:** If the representation supports infinities, the base range is nevertheless restricted to include only the representable finite values, so that 'Base'First and 'Base'Last are always guaranteed to be finite.

6.d **To be honest:** By a "value that can be assigned without overflow" we don't mean to restrict ourselves to values that can be represented exactly. Values between machine representable values can be assigned, but on subsequent reading, a slightly different value might be retrieved, as (partially) determined by the number of digits of precision of the type.

7 [A constrained scalar subtype is one to which a range constraint applies.] The *range* of a constrained scalar subtype is the range associated with the range constraint of the subtype. The *range* of an unconstrained scalar subtype is the base range of its type.

Dynamic Semantics

8 A range is *compatible* with a scalar subtype if and only if it is either a null range or each bound of the range belongs to the range of the subtype. A range_constraint is *compatible* with a scalar subtype if and only if its range is compatible with the subtype.

8.a **Ramification:** Only range_constraints (explicit or implicit) impose conditions on the values of a scalar subtype. The other scalar_constraints, digits_constraints and delta_constraints impose conditions on the subtype denoted by the subtype_mark in a subtype_indication, but don't impose a condition on the values of the subtype being defined. Therefore, a scalar subtype is not called *constrained* if all that applies to it is a digits_constraint. Decimal subtypes are subtle, because a digits_constraint without a range_constraint nevertheless includes an implicit range_constraint.

9 The elaboration of a range_constraint consists of the evaluation of the range. The evaluation of a range determines a lower bound and an upper bound. If simple_expressions are given to specify bounds, the evaluation of the range evaluates these simple_expressions in an arbitrary order, and converts them to the type of the range. If a range_attribute_reference is given, the evaluation of the range consists of the evaluation of the range_attribute_reference.

Attributes 10

For every scalar subtype S, the following attributes are defined: 11

S'First S'First denotes the lower bound of the range of S. The value of this attribute is of the type of S. 12

Ramification: Evaluating S'First never raises Constraint_Error. 12.a

S'Last S'Last denotes the upper bound of the range of S. The value of this attribute is of the type of S. 13

Ramification: Evaluating S'Last never raises Constraint_Error. 13.a

S'Range S'Range is equivalent to the range S'First .. S'Last. 14

S'Base S'Base denotes an unconstrained subtype of the type of S. This unconstrained subtype is called the *base subtype* of the type. 15

S'Min S'Min denotes a function with the following specification: 16

```
function S'Min(Left, Right : S'Base)
  return S'Base
```
17

The function returns the lesser of the values of the two parameters. 18

Discussion: The formal parameter names are italicized because they cannot be used in calls — see 6.4. Such a specification cannot be written by the user because an attribute_reference is not permitted as the designator of a user-defined function, nor can its formal parameters be anonymous. 18.a

S'Max S'Max denotes a function with the following specification: 19

```
function S'Max(Left, Right : S'Base)
  return S'Base
```
20

The function returns the greater of the values of the two parameters. 21

S'Succ S'Succ denotes a function with the following specification: 22

```
function S'Succ(Arg : S'Base)
  return S'Base
```
23

For an enumeration type, the function returns the value whose position number is one more than that of the value of *Arg*; Constraint_Error is raised if there is no such value of the type. For an integer type, the function returns the result of adding one to the value of *Arg*. For a fixed point type, the function returns the result of adding *small* to the value of *Arg*. For a floating point type, the function returns the machine number (as defined in 3.5.7) immediately above the value of *Arg*; Constraint_Error is raised if there is no such machine number. 24

Ramification: S'Succ for a modular integer subtype wraps around if the value of *Arg* is S'Base'Last. S'Succ for a signed integer subtype might raise Constraint_Error if the value of *Arg* is S'Base'Last, or it might return the out-of-base-range value S'Base'Last+1, as is permitted for all predefined numeric operations. 24.a

S'Pred S'Pred denotes a function with the following specification: 25

```
function S'Pred(Arg : S'Base)
  return S'Base
```
26

For an enumeration type, the function returns the value whose position number is one less than that of the value of *Arg*; Constraint_Error is raised if there is no such value of the type. For an integer type, the function returns the result of subtracting one from the value of *Arg*. For a fixed point type, the function returns the result of subtracting *small* from the value of *Arg*. For a floating point type, the function returns the machine number (as defined in 3.5.7) immediately below the value of *Arg*; Constraint_Error is raised if there is no such machine number. 27

Ramification: S'Pred for a modular integer subtype wraps around if the value of *Arg* is S'Base'First. S'Pred for a signed integer subtype might raise Constraint_Error if the value of *Arg* is S'Base'First, or it might return the out-of-base-range value S'Base'First–1, as is permitted for all predefined numeric operations. 27.a

Paragraphs 28 through 37 were moved to 4.10, "Image Attributes".

37.1/5 S'Wide_Wide_Width

{*AI95-00285-01*} {*AI12-0020-1*} S'Wide_Wide_Width denotes the maximum length of a Wide_Wide_String returned by S'Wide_Wide_Image over all values of the subtype S, assuming a default implementation of S'Put_Image. It denotes zero for a subtype that has a null range. Its type is *universal_integer*.

38/5 S'Wide_Width

{*AI12-0020-1*} S'Wide_Width denotes the maximum length of a Wide_String returned by S'Wide_Image over all values of the subtype S, assuming a default implementation of S'Put_Image. It denotes zero for a subtype that has a null range. Its type is *universal_integer*.

39/5 S'Width {*AI12-0020-1*} S'Width denotes the maximum length of a String returned by S'Image over all values of the subtype S, assuming a default implementation of S'Put_Image. It denotes zero for a subtype that has a null range. Its type is *universal_integer*.

39.1/2 S'Wide_Wide_Value

{*AI95-00285-01*} S'Wide_Wide_Value denotes a function with the following specification:

39.2/2
```
function S'Wide_Wide_Value(Arg : Wide_Wide_String)
  return S'Base
```

39.3/2 This function returns a value given an image of the value as a Wide_Wide_String, ignoring any leading or trailing spaces.

39.4/3 {*AI05-0264-1*} For the evaluation of a call on S'Wide_Wide_Value for an enumeration subtype S, if the sequence of characters of the parameter (ignoring leading and trailing spaces) has the syntax of an enumeration literal and if it corresponds to a literal of the type of S (or corresponds to the result of S'Wide_Wide_Image for a nongraphic character of the type), the result is the corresponding enumeration value; otherwise, Constraint_Error is raised.

39.a.1/2 **Discussion:** It's not crystal clear that Range_Check is appropriate here, but it doesn't seem worthwhile to invent a whole new check name just for this weird case, so we decided to lump it in with Range_Check.

39.a.2/2 **To be honest:** {*8652/0096*} {*AI95-00053-01*} A sequence of characters corresponds to the result of S'Wide_Wide_Image if it is the same ignoring case. Thus, the case of an image of a nongraphic character does not matter. For example, Character'Wide_Wide_Value("nul") does not raise Constraint_Error, even though Character'Wide_Wide_Image returns "NUL" for the nul character.

39.5/3 {*AI05-0264-1*} For the evaluation of a call on S'Wide_Wide_Value for an integer subtype S, if the sequence of characters of the parameter (ignoring leading and trailing spaces) has the syntax of an integer literal, with an optional leading sign character (plus or minus for a signed type; only plus for a modular type), and the corresponding numeric value belongs to the base range of the type of S, then that value is the result; otherwise, Constraint_Error is raised.

39.a.3/2 **Discussion:** We considered allowing 'Value to return a representable but out-of-range value without a Constraint_Error. However, we currently require (see 4.9) in an assignment_statement like "X := <numeric_literal>;" that the value of the numeric-literal be in X's base range (at compile time), so it seems unfriendly and confusing to have a different range allowed for 'Value. Furthermore, for modular types, without the requirement for being in the base range, 'Value would have to handle arbitrarily long literals (since overflow never occurs for modular types).

39.6/2 For the evaluation of a call on S'Wide_Wide_Value for a real subtype S, if the sequence of characters of the parameter (ignoring leading and trailing spaces) has the syntax of one of the following:

39.7/2 - numeric_literal

39.8/2 - numeral.[exponent]

39.9/2 - .numeral[exponent]

39.10/2 - base#based_numeral.#[exponent]

- base#.based_numeral#[exponent] 39.11/2

{*AI05-0264-1*} with an optional leading sign character (plus or minus), and if the corresponding numeric value belongs to the base range of the type of S, then that value is the result; otherwise, Constraint_Error is raised. The sign of a zero value is preserved (positive if none has been specified) if S'Signed_Zeros is True. 39.12/3

S'Wide_Value 40

S'Wide_Value denotes a function with the following specification:

```
function S'Wide_Value(Arg : Wide_String)
  return S'Base
```
41

This function returns a value given an image of the value as a Wide_String, ignoring any leading or trailing spaces. 42

{*AI95-00285-01*} {*AI05-0264-1*} {*AI12-0020-1*} For the evaluation of a call on S'Wide_Value for an enumeration subtype S, if the sequence of characters of the parameter (ignoring leading and trailing spaces) has the syntax of an enumeration literal and if it corresponds to a literal of the type of S (or corresponds to the result of S'Wide_Image for a value of the type, assuming a default implementation of S'Put_Image), the result is the corresponding enumeration value; otherwise, Constraint_Error is raised. For a numeric subtype S, the evaluation of a call on S'Wide_Value with *Arg* of type Wide_String is equivalent to a call on S'Wide_Wide_Value for a corresponding *Arg* of type Wide_Wide_String. 43/5

This paragraph was deleted. 43.a/2

This paragraph was deleted.{*8652/0096*} {*AI95-00053-01*} 43.a.1/2

Reason: S'Wide_Value is subtly different from S'Wide_Wide_Value for enumeration subtypes since S'Wide_Image might produce a different sequence of characters than S'Wide_Wide_Image if the enumeration literal uses characters outside of the predefined type Wide_Character. That is why we don't just define S'Wide_Value in terms of S'Wide_Wide_Value for enumeration subtypes. S'Wide_Value and S'Wide_Wide_Value for numeric subtypes yield the same result given the same sequence of characters. 43.b/2

Paragraphs 44 through 51 were moved to Wide_Wide_Value.

S'Value S'Value denotes a function with the following specification: 52

```
function S'Value(Arg : String)
  return S'Base
```
53

This function returns a value given an image of the value as a String, ignoring any leading or trailing spaces. 54

{*AI95-00285-01*} {*AI05-0264-1*} {*AI12-0020-1*} For the evaluation of a call on S'Value for an enumeration subtype S, if the sequence of characters of the parameter (ignoring leading and trailing spaces) has the syntax of an enumeration literal and if it corresponds to a literal of the type of S (or corresponds to the result of S'Image for a value of the type, assuming a default implementation of S'Put_Image), the result is the corresponding enumeration value; otherwise, Constraint_Error is raised. For a numeric subtype S, the evaluation of a call on S'Value with *Arg* of type String is equivalent to a call on S'Wide_Wide_Value for a corresponding *Arg* of type Wide_Wide_String. 55/5

Reason: {*AI95-00285-01*} S'Value is subtly different from S'Wide_Wide_Value for enumeration subtypes; see the discussion under S'Wide_Value. 55.a/2

Implementation Permissions

{*AI95-00285-01*} An implementation may extend the Wide_Wide_Value, [Wide_Value, Value, Wide_Wide_Image, Wide_Image, and Image] attributes of a floating point type to support special values such as infinities and NaNs. 56/2

56.a/2 **Proof:** {*AI95-00285-01*} The permission is really only necessary for Wide_Wide_Value, because Value and Wide_Value are defined in terms of Wide_Wide_Value, and because the behavior of Wide_Wide_Image, Wide_Image, and Image is already unspecified for things like infinities and NaNs.

56.b **Reason:** This is to allow implementations to define full support for IEEE arithmetic. See also the similar permission for Get in A.10.9.

56.1/3 {*AI05-0182-1*} {*AI05-0262-1*} {*AI05-0269-1*} An implementation may extend the Wide_Wide_Value, Wide_Value, and Value attributes of a character type to accept strings of the form "Hex_*hhhhhhhh*" (ignoring case) for any character (not just the ones for which Wide_Wide_Image would produce that form — see 3.5.2), as well as three-character strings of the form "'*X*'", where *X* is any character, including nongraphic characters.

Static Semantics

56.2/3 {*AI05-0228-1*} For a scalar type, the following language-defined representation aspect may be specified with an aspect_specification (see 13.1.1):

56.3/3 Default_Value

This aspect shall be specified by a static expression, and that expression shall be explicit, even if the aspect has a boolean type. Default_Value shall be specified only on a full_type_declaration.

56.c/3 **Reason:** The part about requiring an explicit expression is to disallow omitting the value for this aspect, which would otherwise be allowed by the rules of 13.1.1.

56.d/3 This is a representation aspect in order to disallow specifying it on a derived type that has inherited primitive subprograms; that is necessary as the sizes of **out** parameters could be different whether or not a Default_Value is specified (see 6.4.1).

56.e/3 **Aspect Description for Default_Value:** Default value for a scalar subtype.

56.4/5 {*AI05-0228-1*} {*AI12-0427-1*} If a derived type inherits a boolean Default_Value aspect, the aspect may be specified to have any value for the derived type. If a derived type *T* does not inherit a Default_Value aspect, it shall not specify such an aspect if it inherits a primitive subprogram that has a parameter of type *T* of mode **out**.

56.f/3 **Reason:** This overrides the 13.1.1 rule that says that a boolean aspect with a value True cannot be changed.

56.g/5 {*AI12-0427-1*} The second sentence is to avoid violating the rules specified in 6.4.1 about view conversions of **out** parameters with a specified Default_Value aspect.

Name Resolution Rules

56.5/3 {*AI05-0228-1*} The expected type for the expression specified for the Default_Value aspect is the type defined by the full_type_declaration on which it appears.

57 NOTE 1 The evaluation of S'First or S'Last never raises an exception. If a scalar subtype S has a nonnull range, S'First and S'Last belong to this range. These values can, for example, always be assigned to a variable of subtype S.

57.a **Discussion:** This paragraph addresses an issue that came up with Ada 83, where for fixed point types, the end points of the range specified in the type definition were not necessarily within the base range of the type. However, it was later clarified (and we reconfirm it in 3.5.9, "Fixed Point Types") that the First and Last attributes reflect the true bounds chosen for the type, not the bounds specified in the type definition (which might be outside the ultimately chosen base range).

58/5 NOTE 2 {*AI12-0442-1*} For a subtype of a scalar type, the result delivered by the attributes Succ, Pred, and Value can be outside to the subtype; similarly, the actual parameters of the attributes Succ, Pred, and Image can also be outside the subtype.

59/5 NOTE 3 {*AI95-00285-01*} {*AI12-0020-1*} For any value V (including any nongraphic character) of an enumeration subtype S without a specified Put_Image (see 4.10), S'Value(S'Image(V)) equals V, as do S'Wide_Value(S'Wide_Image(V)) and S'Wide_Wide_Value(S'Wide_Wide_Image(V)). None of these expressions ever raise Constraint_Error.

Examples

Examples of ranges: 60

```
-10 .. 10                                                    61
X .. X + 1
0.0 .. 2.0*Pi
Red .. Green      -- see 3.5.1
1 .. 0            -- a null range
Table'Range       -- a range attribute reference (see 3.6)
```

Examples of range constraints: 62

```
range -999.0 .. +999.0                                       63
range S'First+1 .. S'Last-1
```

Incompatibilities With Ada 83

S'Base is no longer defined for nonscalar types. One conceivable existing use of S'Base for nonscalar types is S'Base'Size where S is a generic formal private type. However, that is not generally useful because the actual subtype corresponding to S might be a constrained array or discriminated type, which would mean that S'Base'Size might very well overflow (for example, S'Base'Size where S is a constrained subtype of String will generally be 8 * (Integer'Last + 1)). For derived discriminated types that are packed, S'Base'Size might not even be well defined if the first subtype is constrained, thereby allowing some amount of normally required "dope" to have been squeezed out in the packing. Hence our conclusion is that S'Base'Size is not generally useful in a generic, and does not justify keeping the attribute Base for nonscalar types just so it can be used as a prefix. 63.a/1

Extensions to Ada 83

The attribute S'Base for a scalar subtype is now permitted anywhere a subtype_mark is permitted. S'Base'First .. S'Base'Last is the base range of the type. Using an attribute_definition_clause, one cannot specify any subtype-specific attributes for the subtype denoted by S'Base (the base subtype). 63.b

The attribute S'Range is now allowed for scalar subtypes. 63.c

The attributes S'Min and S'Max are now defined, and made available for all scalar types. 63.d

The attributes S'Succ, S'Pred, S'Image, S'Value, and S'Width are now defined for real types as well as discrete types. 63.e

Wide_String versions of S'Image and S'Value are defined. These are called S'Wide_Image and S'Wide_Value to avoid introducing ambiguities involving uses of these attributes with string literals. 63.f

Wording Changes from Ada 83

We now use the syntactic category range_attribute_reference since it is now syntactically distinguished from other attribute references. 63.g

The definition of S'Base has been moved here from 3.3.3 since it now applies only to scalar types. 63.h

More explicit rules are provided for nongraphic characters. 63.i

Extensions to Ada 95

{*AI95-00285-01*} The attributes Wide_Wide_Image, Wide_Wide_Value, and Wide_Wide_Width are new. Note that Wide_Image and Wide_Value are now defined in terms of Wide_Wide_Image and Wide_Wide_Value, but the image of types other than characters have not changed. 63.j/2

Wording Changes from Ada 95

{*AI95-00285-01*} The Wide_Image and Wide_Value attributes are now defined in terms of Wide_Wide_Image and Wide_Wide_Value, but the images of numeric types have not changed. 63.k/2

Inconsistencies With Ada 2005

{*AI05-0181-1*} **Correction:** Soft hyphen (code point 173) is nongraphic in ISO/IEC 10646:2011 (and also in the 2003 version of that standard). Thus, we have given it the language-defined name *soft_hyphen*. This changes the result of Character'Image (and all of the related types and Image attributes) for this character, and changes the behavior of Character'Value (and all of the related types and Value attributes) for this character, and (in unusual circumstances), changes the result for Character'Width (and all of the related types and Width attributes). The vast majority of programs won't see any difference, as they are already prepared to handle nongraphic characters. 63.l/3

63.m/3 {*AI05-0182-1*} **Correction:** Added an Implementation Permissions to let Wide_Wide_Value, Wide_Value, and Value accept strings in the form of literals containing nongraphic characters and "Hex_hhhhhhhh" for Latin-1 and graphic characters. These were required to raise Constraint_Error in Ada 2005. Since these attributes aren't very useful, implementations were inconsistent as to whether these were accepted, and since code that would care why the attribute failed seems unlikely, this should not be a problem in practice.

Extensions to Ada 2005

63.n/3 {*AI05-0228-1*} The new aspect Default_Value allows defining implicit initial values (see 3.3.1) for scalar types.

Extensions to Ada 2012

63.o/4 {*AI12-0124-1*} **Corrigendum:** An object can be now used as the prefix of the Image attribute (as well as Wide_Image and Wide_Wide_Image), a convenience feature already present in some implementations.

63.p/5 {*AI12-0020-1*} All of the Image-family attributes have been moved to 4.10.

3.5.1 Enumeration Types

1 [An enumeration_type_definition defines an enumeration type.]

Syntax

2 enumeration_type_definition ::=
(enumeration_literal_specification {, enumeration_literal_specification})

3 enumeration_literal_specification ::=
defining_identifier | defining_character_literal

4 defining_character_literal ::= character_literal

Legality Rules

5/3 {*AI05-0227-1*} {*AI05-0299-1*} The defining_identifiers in upper case [and the defining_character_literals] listed in an enumeration_type_definition shall be distinct.

5.a/3 **Proof:** {*AI05-0227-1*} For character literals, this is a ramification of the normal disallowance of homographs explicitly declared immediately in the same declarative region.

5.b/3 **Reason:** {*AI05-0227-1*} To ease implementation of the attribute Wide_Wide_Value, we require that all enumeration literals have distinct images.

Static Semantics

6/3 {*AI05-0006-1*} Each enumeration_literal_specification is the explicit declaration of the corresponding *enumeration literal*: it declares a parameterless function, whose defining name is the defining_identifier or defining_character_literal, and whose result subtype is the base subtype of the enumeration type.

6.a **Reason:** This rule defines the profile of the enumeration literal, which is used in the various types of conformance.

6.b **Ramification:** The parameterless function associated with an enumeration literal is fully defined by the enumeration_type_definition; a body is not permitted for it, and it never fails the Elaboration_Check when called.

6.c/3 **Discussion:** {*AI05-0006-1*} The result subtype is primarily a concern when an enumeration literal is used as the expression of a case statement, due to the full coverage requirement based on the nominal subtype.

7 Each enumeration literal corresponds to a distinct value of the enumeration type, and to a distinct position number. The position number of the value of the first listed enumeration literal is zero; the position number of the value of each subsequent enumeration literal is one more than that of its predecessor in the list.

8 [The predefined order relations between values of the enumeration type follow the order of corresponding position numbers.]

[If the same defining_identifier or defining_character_literal is specified in more than one enumeration_type_definition, the corresponding enumeration literals are said to be *overloaded*. At any place where an overloaded enumeration literal occurs in the text of a program, the type of the enumeration literal has to be determinable from the context (see 8.6).] 9

Dynamic Semantics

The elaboration of an enumeration_type_definition creates the enumeration type and its first subtype, which is constrained to the base range of the type. 10

Ramification: The first subtype of a discrete type is always constrained, except in the case of a derived type whose parent subtype is Whatever'Base. 10.a

When called, the parameterless function associated with an enumeration literal returns the corresponding value of the enumeration type. 11

NOTE If an enumeration literal occurs in a context that does not otherwise suffice to determine the type of the literal, then qualification by the name of the enumeration type is one way to resolve the ambiguity (see 4.7). 12

Examples

Examples of enumeration types and subtypes: 13

```
{AI12-0386-1} type Day        is (Mon, Tue, Wed, Thu, Fri, Sat, Sun);                14/5
type Month_Name is (January, February, March, April, May, June, July,
                    August, September, October, November, December);
type Suit       is (Clubs, Diamonds, Hearts, Spades);
type Gender     is (M, F);
type Level      is (Low, Medium, Urgent);
type Color      is (White, Red, Yellow, Green, Blue, Brown, Black);
type Light      is (Red, Amber, Green); -- Red and Green are overloaded

type Hexa       is ('A', 'B', 'C', 'D', 'E', 'F');                                    15
type Mixed      is ('A', 'B', '*', B, None, '?', '%');

subtype Weekday is Day   range Mon .. Fri;                                            16
subtype Major   is Suit  range Hearts .. Spades;
subtype Rainbow is Color range Red .. Blue;
                               -- the Color Red, not the Light
```

Wording Changes from Ada 83

The syntax rule for defining_character_literal is new. It is used for the defining occurrence of a character_literal, analogously to defining_identifier. Usage occurrences use the name or selector_name syntactic categories. 16.a

We emphasize the fact that an enumeration literal denotes a function, which is called to produce a value. 16.b

Incompatibilities With Ada 2005

{*AI05-0227-1*} **Correction:** Required that all enumeration literals in a type have distinct images; this might not be the case since upper case conversion can map distinct characters to the same upper case character. This can only happen for identifiers using Unicode characters first allowed by Ada 2005; moreover, the original definition of Ada 2005 was confused and appeared to require inconsistent results from the Image attribute, so implementations that allowed problematic cases are rare; the problematic cases are very rare; so it is expected that this change would only affect test programs. 16.c/3

Wording Changes from Ada 2005

{*AI05-0006-1*} **Correction:** Defined the result subtype of an enumeration literal to close a minor language hole. 16.d/3

3.5.2 Character Types

Static Semantics

An enumeration type is said to be a *character type* if at least one of its enumeration literals is a character_literal. 1

2/5 {*AI95-00285-01*} {*AI05-0181-1*} {*AI05-0262-1*} {*AI05-0266-1*} {*AI12-0263-1*} {*AI12-0450-1*} The predefined type Character is a character type whose values correspond to the 256 code points of Row 00 (also known as Latin-1) of the ISO/IEC 10646:2020 Basic Multilingual Plane (BMP). Each of the graphic characters of Row 00 of the BMP has a corresponding character_literal in Character. Each of the nongraphic characters of Row 00 has a corresponding language-defined name, which is not usable as an enumeration literal, but which is usable with the attributes Image, Wide_Image, Wide_Wide_Image, Value, Wide_Value, and Wide_Wide_Value; these names are given in the definition of type Character in A.1, "The Package Standard", but are set in *italics*.

2.a/5 **Discussion:** {*AI05-0262-1*} {*AI12-0263-1*} {*AI12-0450-1*} *Code point* is defined in ISO/IEC 10646:2020.

3/5 {*AI95-00285-01*} {*AI05-0262-1*} {*AI12-0263-1*} {*AI12-0450-1*} The predefined type Wide_Character is a character type whose values correspond to the 65536 code points of the ISO/IEC 10646:2020 Basic Multilingual Plane (BMP). Each of the graphic characters of the BMP has a corresponding character_literal in Wide_Character. The first 256 values of Wide_Character have the same character_literal or language-defined name as defined for Character. Each of the graphic_characters has a corresponding character_literal.

4/5 {*AI95-00285-01*} {*AI05-0262-1*} {*AI12-0263-1*} {*AI12-0450-1*} The predefined type Wide_Wide_Character is a character type whose values correspond to the 2147483648 code points of the ISO/IEC 10646:2020 character set. Each of the graphic_characters has a corresponding character_literal in Wide_Wide_Character. The first 65536 values of Wide_Wide_Character have the same character_literal or language-defined name as defined for Wide_Character.

5/3 {*AI95-00285-01*} {*AI05-0262-1*} The characters whose code point is larger than 16#FF# and which are not graphic_characters have language-defined names which are formed by appending to the string "Hex_" the representation of their code point in hexadecimal as eight extended digits. As with other language-defined names, these names are usable only with the attributes (Wide_)Wide_Image and (Wide_)Wide_Value; they are not usable as enumeration literals.

5.a/2 **Reason:** {*AI95-00285-01*} The language-defined names are not usable as enumeration literals to avoid "polluting" the name space. Since Wide_Character and Wide_Wide_Character are defined in Standard, if the language-defined names were usable as enumeration literals, they would hide other nonoverloadable declarations with the same names in **use**-d packages.

Original Paragraphs 4 and 5 were deleted.

8 NOTE 1 The language-defined library package Characters.Latin_1 (see A.3.3) includes the declaration of constants denoting control characters, lower case characters, and special characters of the predefined type Character.

8.a **To be honest:** The package ASCII does the same, but only for the first 128 characters of Character. Hence, it is an obsolescent package, and we no longer mention it here.

9/5 NOTE 2 {*AI05-0299-1*} {*AI12-0449-1*} A conventional character set such as *EBCDIC* can be declared as a character type; the internal codes of the characters can be specified by an enumeration_representation_clause as explained in 13.4.

Examples

10 *Example of a character type:*

11
```
type Roman_Digit is ('I', 'V', 'X', 'L', 'C', 'D', 'M');
```

Inconsistencies With Ada 83

11.a The declaration of Wide_Character in package Standard hides use-visible declarations with the same defining identifier. In the unlikely event that an Ada 83 program had depended on such a use-visible declaration, and the program remains legal after the substitution of Standard.Wide_Character, the meaning of the program will be different.

Incompatibilities With Ada 83

11.b The presence of Wide_Character in package Standard means that an expression such as

11.c
```
'a' = 'b'
```

is ambiguous in Ada 95, whereas in Ada 83 both literals could be resolved to be of type Character. 11.d

The change in visibility rules (see 4.2) for character literals means that additional qualification might be necessary to resolve expressions involving overloaded subprograms and character literals. 11.e

Extensions to Ada 83

The type Character has been extended to have 256 positions, and the type Wide_Character has been added. Note that this change was already approved by the ARG for Ada 83 conforming compilers. 11.f

The rules for referencing character literals are changed (see 4.2), so that the declaration of the character type need not be directly visible to use its literals, similar to **null** and string literals. Context is used to resolve their type. 11.g

Inconsistencies With Ada 95

{*AI95-00285-01*} Ada 95 defined most characters in Wide_Character to be graphic characters, while Ada 2005 uses the categorizations from ISO-10646:2003. It also provides language-defined names for all nongraphic characters. That means that in Ada 2005, Wide_Character'Wide_Value will raise Constraint_Error for a string representing a character_literal of a nongraphic character, while Ada 95 would have accepted it. Similarly, the result of Wide_Character'Wide_Image will change for such nongraphic characters. 11.h/2

{*AI95-00395-01*} {*AI05-0005-1*} {*AI05-0262-1*} The language-defined names FFFE and FFFF were replaced by a consistent set of language-defined names for all nongraphic characters with code points greater than 16#FF#. That means that in Ada 2005, Wide_Character'Wide_Value("FFFE") will raise Constraint_Error while Ada 95 would have accepted it. Similarly, the result of Wide_Character'Wide_Image will change for the position numbers 16#FFFE# and 16#FFFF#. It is very unlikely that this will matter in practice, as these names do not represent usable characters. 11.i/3

{*AI95-00285-01*} {*AI95-00395-01*} Because of the previously mentioned changes to the Wide_Character'Wide_Image of various character values, the value of attribute Wide_Width will change for some subtypes of Wide_Character. However, the new language-defined names were chosen so that the value of Wide_Character'Wide_Width itself does not change. 11.j/2

{*AI95-00285-01*} The declaration of Wide_Wide_Character in package Standard hides use-visible declarations with the same defining identifier. In the (very) unlikely event that an Ada 95 program had depended on such a use-visible declaration, and the program remains legal after the substitution of Standard.Wide_Wide_Character, the meaning of the program will be different. 11.k/2

Extensions to Ada 95

{*AI95-00285-01*} The type Wide_Wide_Character is new. 11.l/2

Wording Changes from Ada 95

{*AI95-00285-01*} Characters are now defined in terms of the entire ISO/IEC 10646:2003 character set. 11.m/2

{*AI95-00285-01*} {*AI05-0248-1*} We dropped the Implementation Advice for nonstandard interpretation of character sets; an implementation can do what it wants in a nonstandard mode, so there isn't much point to any advice. 11.n/3

Wording Changes from Ada 2005

{*AI05-0181-1*} **Correction:** Removed the position numbers of nongraphic characters from the text, as it is wrong and thus misleading. 11.o/3

{*AI05-0262-1*} Changed "code position" to "code point" consistently throughout the standard, as ISO/IEC 10646:2011 prefers "code point" and we are referring to the definition in that Standard. This change also reduces confusion between "code point" and "position number"; while these have the same values for the predefined character types, there is no required relationship for other character types. 11.p/3

3.5.3 Boolean Types

Static Semantics

There is a predefined enumeration type named Boolean, [declared in the visible part of package Standard]. It has the two enumeration literals False and True ordered with the relation False < True. Any descendant of the predefined type Boolean is called a *boolean* type. 1

Implementation Note: An implementation is not required to support enumeration representation clauses on boolean types that impose an unacceptable implementation burden. See 13.4, “Enumeration Representation Clauses”. However, it is generally straightforward to support representations where False is zero and True is 2**n – 1 for some n. 1.a

3.5.4 Integer Types

1 An integer_type_definition defines an integer type; it defines either a *signed* integer type, or a *modular* integer type. The base range of a signed integer type includes at least the values of the specified range. A modular type is an integer type with all arithmetic modulo a specified positive *modulus*; such a type corresponds to an unsigned type with wrap-around semantics.

Syntax

2 integer_type_definition ::=
signed_integer_type_definition | modular_type_definition

3 signed_integer_type_definition ::=
range *static*_simple_expression .. *static*_simple_expression

3.a **Discussion:** We don't call this a range_constraint, because it is rather different — not only is it required to be static, but the associated overload resolution rules are different than for normal range constraints. A similar comment applies to real_range_specification. This used to be integer_range_specification but when we added support for modular types, it seemed overkill to have three levels of syntax rules, and just calling these signed_integer_range_specification and modular_range_specification loses the fact that they are defining different classes of types, which is important for the generic type matching rules.

4 modular_type_definition ::= **mod** *static*_expression

Name Resolution Rules

5/5 {*AI12-0444-1*} Each simple_expression in a signed_integer_type_definition is expected to be of any integer type; they can be of different integer types. The expression in a modular_type_definition is likewise expected to be of any integer type.

Legality Rules

6 The simple_expressions of a signed_integer_type_definition shall be static, and their values shall be in the range System.Min_Int .. System.Max_Int.

7 The expression of a modular_type_definition shall be static, and its value (the *modulus*) shall be positive, and shall be no greater than System.Max_Binary_Modulus if a power of 2, or no greater than System.Max_Nonbinary_Modulus if not.

7.a **Reason:** For a 2's-complement machine, supporting nonbinary moduli greater than System.Max_Int can be quite difficult, whereas essentially any binary moduli are straightforward to support, up to 2*System.Max_Int+2, so this justifies having two separate limits.

Static Semantics

8 The set of values for a signed integer type is the (infinite) set of mathematical integers[, though only values of the base range of the type are fully supported for run-time operations]. The set of values for a modular integer type are the values from 0 to one less than the modulus, inclusive.

9 A signed_integer_type_definition defines an integer type whose base range includes at least the values of the simple_expressions and is symmetric about zero, excepting possibly an extra negative value. A signed_integer_type_definition also defines a constrained first subtype of the type, with a range whose bounds are given by the values of the simple_expressions, converted to the type being defined.

9.a/2 **Implementation Note:** {*AI95-00114-01*} The base range of a signed integer type might be much larger than is necessary to satisfy the above requirements.

9.a.1/1 **To be honest:** The conversion mentioned above is not an *implicit subtype conversion* (which is something that happens at overload resolution, see 4.6), although it happens implicitly. Therefore, the freezing rules are not invoked on the type (which is important so that representation items can be given for the type).

A modular_type_definition defines a modular type whose base range is from zero to one less than the given modulus. A modular_type_definition also defines a constrained first subtype of the type with a range that is the same as the base range of the type. 10

There is a predefined signed integer subtype named Integer[, declared in the visible part of package Standard]. It is constrained to the base range of its type. 11

Reason: Integer is a constrained subtype, rather than an unconstrained subtype. This means that on assignment to an object of subtype Integer, a range check is required. On the other hand, an object of subtype Integer'Base is unconstrained, and no range check (only overflow check) is required on assignment. For example, if the object is held in an extended-length register, its value might be outside of Integer'First .. Integer'Last. All parameter and result subtypes of the predefined integer operators are of such unconstrained subtypes, allowing extended-length registers to be used as operands or for the result. In an earlier version of Ada 95, Integer was unconstrained. However, the fact that certain Constraint_Errors might be omitted or appear elsewhere was felt to be an undesirable upward inconsistency in this case. Note that for Float, the opposite conclusion was reached, partly because of the high cost of performing range checks when not actually necessary. Objects of subtype Float are unconstrained, and no range checks, only overflow checks, are performed for them. 11.a

Integer has two predefined subtypes, [declared in the visible part of package Standard:] 12

```
subtype Natural  is Integer range 0 .. Integer'Last;
subtype Positive is Integer range 1 .. Integer'Last;
```
13

A type defined by an integer_type_definition is implicitly derived from *root_integer*, an anonymous predefined (specific) integer type, whose base range is System.Min_Int .. System.Max_Int. However, the base range of the new type is not inherited from *root_integer*, but is instead determined by the range or modulus specified by the integer_type_definition. [Integer literals are all of the type *universal_integer*, the universal type (see 3.4.1) for the class rooted at *root_integer*, allowing their use with the operations of any integer type.] 14

Discussion: This implicit derivation is not considered exactly equivalent to explicit derivation via a derived_type_definition. In particular, integer types defined via a derived_type_definition inherit their base range from their parent type. A type defined by an integer_type_definition does not necessarily inherit its base range from *root_integer*. It is not specified whether the implicit derivation from *root_integer* is direct or indirect, not that it really matters. All we want is for all integer types to be descendants of *root_integer*. 14.a

{*8652/0099*} {*AI95-00152-01*} Note that this derivation does not imply any inheritance of subprograms. Subprograms are inherited only for types derived by a derived_type_definition (see 3.4), or a private_extension_declaration (see 7.3, 7.3.1, and 12.5.1). 14.a.1/1

Implementation Note: It is the intent that even nonstandard integer types (see below) will be descendants of *root_integer*, even though they might have a base range that exceeds that of *root_integer*. This causes no problem for static calculations, which are performed without range restrictions (see 4.9). However for run-time calculations, it is possible that Constraint_Error might be raised when using an operator of *root_integer* on the result of 'Val applied to a value of a nonstandard integer type. 14.b

The *position number* of an integer value is equal to the value. 15

{*AI95-00340-01*} For every modular subtype S, the following attributes are defined: 16/2

S'Mod {*AI95-00340-01*} S'Mod denotes a function with the following specification: 16.1/2

```
function S'Mod (Arg : universal_integer)
  return S'Base
```
16.2/2

This function returns *Arg* **mod** S'Modulus, as a value of the type of S. 16.3/2

S'Modulus S'Modulus yields the modulus of the type of S, as a value of the type *universal_integer*. 17

Dynamic Semantics

The elaboration of an integer_type_definition creates the integer type and its first subtype. 18

For a modular type, if the result of the execution of a predefined operator (see 4.5) is outside the base range of the type, the result is reduced modulo the modulus of the type to a value that is within the base range of the type. 19

20 For a signed integer type, the exception Constraint_Error is raised by the execution of an operation that cannot deliver the correct result because it is outside the base range of the type. [For any integer type, Constraint_Error is raised by the operators "/", "**rem**", and "**mod**" if the right operand is zero.]

Implementation Requirements

21 In an implementation, the range of Integer shall include the range –2**15+1 .. +2**15–1.

22 If Long_Integer is predefined for an implementation, then its range shall include the range –2**31+1 .. +2**31–1.

23 System.Max_Binary_Modulus shall be at least 2**16.

Implementation Permissions

24/5 {*AI12-0444-1*} For the execution of a predefined operation of a signed integer type, it is optional to raise Constraint_Error if the result is outside the base range of the type, so long as the correct result is produced.

24.a **Discussion:** Constraint_Error is never raised for operations on modular types, except for divide-by-zero (and **rem**/**mod**-by-zero).

25 An implementation may provide additional predefined signed integer types[, declared in the visible part of Standard], whose first subtypes have names of the form Short_Integer, Long_Integer, Short_Short_Integer, Long_Long_Integer, etc. Different predefined integer types are allowed to have the same base range. However, the range of Integer should be no wider than that of Long_Integer. Similarly, the range of Short_Integer (if provided) should be no wider than Integer. Corresponding recommendations apply to any other predefined integer types. An implementation may support base ranges for which there is no corresponding named integer type. The range of each first subtype should be the base range of its type.

25.a/5 {*AI12-0444-1*} **Implementation defined:** The predefined integer types declared in Standard.

26/5 {*AI12-0444-1*} An implementation may provide *nonstandard integer types*, descendants of *root_integer* that are declared outside of the specification of package Standard, which may have different characteristics than a type defined by an integer_type_definition. For example, a nonstandard integer type can have an asymmetric base range or it can be disallowed as an array or loop index (a very long integer). Any type descended from a nonstandard integer type is also nonstandard. An implementation may place arbitrary restrictions on the use of such types; it is implementation defined whether operators that are predefined for "any integer type" are defined for a particular nonstandard integer type. [In any case, such types are not permitted as explicit_generic_actual_parameters for formal scalar types — see 12.5.2.]

26.a **Implementation defined:** Any nonstandard integer types and the operators defined for them.

27 For a one's complement machine, the high bound of the base range of a modular type whose modulus is one less than a power of 2 may be equal to the modulus, rather than one less than the modulus. It is implementation defined for which powers of 2, if any, this permission is exercised.

27.1/1 {*8652/0003*} {*AI95-00095-01*} For a one's complement machine, implementations may support nonbinary modulus values greater than System.Max_Nonbinary_Modulus. It is implementation defined which specific values greater than System.Max_Nonbinary_Modulus, if any, are supported.

27.a.1/1 **Reason:** On a one's complement machine, the natural full word type would have a modulus of 2**Word_Size–1. However, we would want to allow the all-ones bit pattern (which represents negative zero as a number) in logical operations. These permissions are intended to allow that and the natural modulus value without burdening implementations with supporting expensive modulus values.

Implementation Advice

An implementation should support Long_Integer in addition to Integer if the target machine supports 32-bit (or longer) arithmetic. No other named integer subtypes are recommended for package Standard. Instead, appropriate named integer subtypes should be provided in the library package Interfaces (see B.2). 28

Implementation Advice: Long_Integer should be declared in Standard if the target supports 32-bit arithmetic. No other named integer subtypes should be declared in Standard. 28.a.1/2

Implementation Note: To promote portability, implementations should explicitly declare the integer (sub)types Integer and Long_Integer in Standard, and leave other predefined integer types anonymous. For implementations that already support Byte_Integer, etc., upward compatibility argues for keeping such declarations in Standard during the transition period, but perhaps generating a warning on use. A separate package Interfaces in the predefined environment is available for pre-declaring types such as Integer_8, Integer_16, etc. See B.2. In any case, if the user declares a subtype (first or not) whose range fits in, for example, a byte, the implementation can store variables of the subtype in a single byte, even if the base range of the type is wider. 28.a

An implementation for a two's complement machine should support modular types with a binary modulus up to System.Max_Int*2+2. An implementation should support a nonbinary modulus up to Integer'Last. 29

Implementation Advice: For a two's complement target, modular types with a binary modulus up to System.Max_Int*2+2 should be supported. A nonbinary modulus up to Integer'Last should be supported. 29.a.1/2

Reason: Modular types provide bit-wise "**and**", "**or**", "**xor**", and "**not**" operations. It is important for systems programming that these be available for all integer types of the target hardware. 29.a

Ramification: Note that on a one's complement machine, the largest supported modular type would normally have a nonbinary modulus. On a two's complement machine, the largest supported modular type would normally have a binary modulus. 29.b

Implementation Note: Supporting a nonbinary modulus greater than Integer'Last can impose an undesirable implementation burden on some machines. 29.c

NOTE 1 Integer literals are of the anonymous predefined integer type *universal_integer*. Other integer types have no literals. However, the overload resolution rules (see 8.6, “The Context of Overload Resolution”) allow expressions of the type *universal_integer* whenever an integer type is expected. 30

NOTE 2 The same arithmetic operators are predefined for all signed integer types defined by a signed_integer_type_definition (see 4.5, “Operators and Expression Evaluation”). For modular types, these same operators are predefined, plus bit-wise logical operators (**and**, **or**, **xor**, and **not**). In addition, for the unsigned types declared in the language-defined package Interfaces (see B.2), functions are defined that provide bit-wise shifting and rotating. 31

NOTE 3 Modular types match a generic_formal_parameter_declaration of the form "**type** T **is mod** <>;"; signed integer types match "**type** T **is range** <>;" (see 12.5.2). 32

Examples

Examples of integer types and subtypes: 33

```
type Page_Num  is range 1 .. 2_000;                                    34
type Line_Size is range 1 .. Max_Line_Size;

subtype Small_Int   is Integer   range -10 .. 10;                      35
subtype Column_Ptr  is Line_Size range 1 .. 10;
subtype Buffer_Size is Integer   range 0 .. Max;

type Byte        is mod 256; -- an unsigned byte                       36
type Hash_Index  is mod 97;  -- modulus is prime
```

Extensions to Ada 83

An implementation is allowed to support any number of distinct base ranges for integer types, even if fewer integer types are explicitly declared in Standard. 36.a

Modular (unsigned, wrap-around) types are new. 36.b

Wording Changes from Ada 83

Ada 83's integer types are now called "signed" integer types, to contrast them with "modular" integer types. 36.c

36.d Standard.Integer, Standard.Long_Integer, etc., denote constrained subtypes of predefined integer types, consistent with the Ada 95 model that only subtypes have names.

36.e We now impose minimum requirements on the base range of Integer and Long_Integer.

36.f We no longer explain integer type definition in terms of an equivalence to a normal type derivation, except to say that all integer types are by definition implicitly derived from *root_integer*. This is for various reasons.

36.g First of all, the equivalence with a type derivation and a subtype declaration was not perfect, and was the source of various AIs (for example, is the conversion of the bounds static? Is a numeric type a derived type with respect to other rules of the language?)

36.h Secondly, we don't want to require that every integer size supported shall have a corresponding named type in Standard. Adding named types to Standard creates nonportabilities.

36.i Thirdly, we don't want the set of types that match a formal derived type "type T is new Integer;" to depend on the particular underlying integer representation chosen to implement a given user-defined integer type. Hence, we would have needed anonymous integer types as parent types for the implicit derivation anyway. We have simply chosen to identify only one anonymous integer type — *root_integer*, and stated that every integer type is derived from it.

36.j Finally, the “fiction” that there were distinct preexisting predefined types for every supported representation breaks down for fixed point with arbitrary smalls, and was never exploited for enumeration types, array types, etc. Hence, there seems little benefit to pushing an explicit equivalence between integer type definition and normal type derivation.

Extensions to Ada 95

36.k/2 {*AI95-00340-01*} The Mod attribute is new. It eases mixing of signed and unsigned values in an expression, which can be difficult as there may be no type which can contain all of the values of both of the types involved.

Wording Changes from Ada 95

36.l/2 {*8652/0003*} {*AI95-00095-01*} **Corrigendum:** Added additional permissions for modular types on one's complement machines.

3.5.5 Operations of Discrete Types

Static Semantics

1 For every discrete subtype S, the following attributes are defined:

2 S'Pos S'Pos denotes a function with the following specification:

3
```
function S'Pos(Arg : S'Base)
  return universal_integer
```

4 This function returns the position number of the value of *Arg*, as a value of type *universal_integer*.

5 S'Val S'Val denotes a function with the following specification:

6
```
function S'Val(Arg : universal_integer)
  return S'Base
```

7 This function returns a value of the type of S whose position number equals the value of *Arg*. For the evaluation of a call on S'Val, if there is no value in the base range of its type with the given position number, Constraint_Error is raised.

7.a **Ramification:** By the overload resolution rules, a formal parameter of type *universal_integer* allows an actual parameter of any integer type.

7.b **Reason:** We considered allowing S'Val for a signed integer subtype S to return an out-of-range value, but since checks were required for enumeration and modular types anyway, the allowance didn't seem worth the complexity of the rule.

7.1/4 {*AI05-0297-1*} {*AI12-0071-1*} For every static discrete subtype S for which there exists at least one value belonging to S that satisfies the predicates of S, the following attributes are defined:

7.2/4 S'First_Valid

{*AI05-0297-1*} {*AI12-0071-1*} S'First_Valid denotes the smallest value that belongs to S and satisfies the predicates of S. The value of this attribute is of the type of S.

S'Last_Valid 7.3/4

{*AI05-0297-1*} {*AI12-0071-1*} S'Last_Valid denotes the largest value that belongs to S and satisfies the predicates of S. The value of this attribute is of the type of S.

{*AI05-0297-1*} [First_Valid and Last_Valid attribute_references are always static expressions. Any explicit predicate of S can only have been specified by a Static_Predicate aspect.] 7.4/3

Proof: An attribute_reference is static if the prefix is a static subtype (see 4.9), (true by definition) and any arguments are static (there are none). Similarly, a dynamic predicate always makes a subtype nonstatic. QED. 7.c/3

Reason: We require there to be at least one value so that these are always values of the subtype. (This sidesteps the question of what to return for a subtype with no values.) 7.d/3

Discussion: These attributes are intended primarily for use in the case where the Static_Predicate aspect of S has been specified; First and Last are equivalent if these are allowed and there is no predicate. 7.e/3

Implementation Advice

For the evaluation of a call on S'Pos for an enumeration subtype, if the value of the operand does not correspond to the internal code for any enumeration literal of its type [(perhaps due to an uninitialized variable)], then the implementation should raise Program_Error. This is particularly important for enumeration types with noncontiguous internal codes specified by an enumeration_representation_-clause. 8

Implementation Advice: Program_Error should be raised for the evaluation of S'Pos for an enumeration type, if the value of the operand does not correspond to the internal code for any enumeration literal of the type. 8.a.1/2

Reason: We say Program_Error here, rather than Constraint_Error, because the main reason for such values is uninitialized variables, and the normal way to indicate such a use (if detected) is to raise Program_Error. (Other reasons would involve the misuse of low-level features such as Unchecked_Conversion.) 8.a

NOTE 1 Indexing and loop iteration use values of discrete types. 9

NOTE 2 The predefined operations of a discrete type include the assignment operation, qualification, the membership tests, and the relational operators; for a boolean type they include the short-circuit control forms and the logical operators; for an integer type they include type conversion to and from other numeric types, as well as the binary and unary adding operators – and +, the multiplying operators, the unary operator **abs**, and the exponentiation operator. The assignment operation is described in 5.2. The other predefined operations are described in Clause 4. 10

NOTE 3 As for all types, objects of a discrete type have Size and Address attributes (see 13.3). 11

NOTE 4 {*AI12-0442-1*} For a subtype of a discrete type, the result delivered by the attribute Val can be outside the subtype; similarly, the actual parameter of the attribute Pos can also be outside the subtype. The following relations are satisfied (in the absence of an exception) by these attributes: 12/5

```
S'Val(S'Pos(X)) = X
S'Pos(S'Val(N)) = N
```
13

Examples

Examples of attributes of discrete subtypes: 14

{*AI12-0449-1*} *-- For the types and subtypes declared in 3.5.1 the following hold:* 15/5

```
--  Color'First   = White,   Color'Last   = Black
--  Rainbow'First = Red,     Rainbow'Last = Blue
```
16

```
--  Color'Succ(Blue) = Rainbow'Succ(Blue) = Brown
--  Color'Pos(Blue)  = Rainbow'Pos(Blue)  = 4
--  Color'Val(0)     = Rainbow'Val(0)     = White
```
17

Extensions to Ada 83

The attributes S'Succ, S'Pred, S'Width, S'Image, and S'Value have been generalized to apply to real types as well (see 3.5, “Scalar Types”). 17.a

Extensions to Ada 2005

{*AI05-0297-1*} The attributes S'First_Valid and S'Last_Valid are new. 17.b/3

Wording Changes from Ada 2012

17.c/4 {*AI12-0071-1*} **Corrigendum:** Updated wording of the attributes S'First_Valid and S'Last_Valid to use the new term "satisfies the predicates" (see 3.2.4).

3.5.6 Real Types

1 Real types provide approximations to the real numbers, with relative bounds on errors for floating point types, and with absolute bounds for fixed point types.

Syntax

2 real_type_definition ::=
floating_point_definition | fixed_point_definition

Static Semantics

3 A type defined by a real_type_definition is implicitly derived from *root_real*, an anonymous predefined (specific) real type. [Hence, all real types, whether floating point or fixed point, are in the derivation class rooted at *root_real*.]

3.a **Ramification:** It is not specified whether the derivation from *root_real* is direct or indirect, not that it really matters. All we want is for all real types to be descendants of *root_real*.

3.a.1/1 {*8652/0099*} {*AI95-00152-01*} Note that this derivation does not imply any inheritance of subprograms. Subprograms are inherited only for types derived by a derived_type_definition (see 3.4), or a private_extension_declaration (see 7.3, 7.3.1, and 12.5.1).

4 [Real literals are all of the type *universal_real*, the universal type (see 3.4.1) for the class rooted at *root_real*, allowing their use with the operations of any real type. Certain multiplying operators have a result type of *universal_fixed* (see 4.5.5), the universal type for the class of fixed point types, allowing the result of the multiplication or division to be used where any specific fixed point type is expected.]

Dynamic Semantics

5 The elaboration of a real_type_definition consists of the elaboration of the floating_point_definition or the fixed_point_definition.

Implementation Requirements

6 An implementation shall perform the run-time evaluation of a use of a predefined operator of *root_real* with an accuracy at least as great as that of any floating point type definable by a floating_point_definition.

6.a **Ramification:** Static calculations using the operators of *root_real* are exact, as for all static calculations. See 4.9.

6.b **Implementation Note:** The Digits attribute of the type used to represent *root_real* at run time is at least as great as that of any other floating point type defined by a floating_point_definition, and its safe range includes that of any such floating point type with the same Digits attribute. On some machines, there might be real types with less accuracy but a wider range, and hence run-time calculations with *root_real* might not be able to accommodate all values that can be represented at run time in such floating point or fixed point types.

Implementation Permissions

7/5 {*AI95-00114-01*} {*AI12-0426-1*} {*AI12-0444-1*} [For the execution of a predefined operation of a real type, it is optional to raise Constraint_Error if the result is outside the base range of the type, so long as the correct result is produced, or the Machine_Overflows attribute of the type is False (see G.2.1).]

8/5 {*AI12-0426-1*} {*AI12-0444-1*} An implementation may provide *nonstandard real types*, descendants of *root_real* that are declared outside of the specification of package Standard, which may have different characteristics than a type defined by a real_type_definition. For example, a nonstandard real type can

have an asymmetric or unsigned base range, or its predefined operations can wrap around or "saturate" rather than overflow (modular or saturating arithmetic), or it can have a different accuracy model than is standard (see G.2.1). Any type descended from a nonstandard real type is also nonstandard. An implementation may place arbitrary restrictions on the use of such types; it is implementation defined whether operators that are predefined for "any real type" are defined for a particular nonstandard real type. [In any case, such types are not permitted as explicit_generic_actual_parameters for formal scalar types — see 12.5.2.]

Implementation defined: Any nonstandard real types and the operators defined for them. 8.a

NOTE As stated, real literals are of the anonymous predefined real type *universal_real*. Other real types have no literals. However, the overload resolution rules (see 8.6) allow expressions of the type *universal_real* whenever a real type is expected. 9

Wording Changes from Ada 83

The syntax rule for real_type_definition is modified to use the new syntactic categories floating_point_definition and fixed_point_definition, instead of floating_point_constraint and fixed_point_constraint, because the semantics of a type definition are significantly different than the semantics of a constraint. 9.a

All discussion of model numbers, safe ranges, and machine numbers is moved to 3.5.7, 3.5.8, and G.2. Values of a fixed point type are now described as being multiples of the *small* of the fixed point type, and we have no need for model numbers, safe ranges, etc. for fixed point types. 9.b

3.5.7 Floating Point Types

For floating point types, the error bound is specified as a relative precision by giving the required minimum number of significant decimal digits. 1

Syntax

floating_point_definition ::= 2
  digits *static*_expression [real_range_specification]

real_range_specification ::= 3
  range *static*_simple_expression .. *static*_simple_expression

Name Resolution Rules

The *requested decimal precision*, which is the minimum number of significant decimal digits required for the floating point type, is specified by the value of the expression given after the reserved word **digits**. This expression is expected to be of any integer type. 4

{*AI12-0444-1*} Each simple_expression of a real_range_specification is expected to be of any real type[; the types can be different]. 5/5

Legality Rules

The requested decimal precision shall be specified by a static expression whose value is positive and no greater than System.Max_Base_Digits. Each simple_expression of a real_range_specification shall also be static. If the real_range_specification is omitted, the requested decimal precision shall be no greater than System.Max_Digits. 6

Reason: We have added Max_Base_Digits to package System. It corresponds to the requested decimal precision of *root_real*. System.Max_Digits corresponds to the maximum value for Digits that may be specified in the absence of a real_range_specification, for upward compatibility. These might not be the same if *root_real* has a base range that does not include ± 10.0**(4*Max_Base_Digits). 6.a

A floating_point_definition is illegal if the implementation does not support a floating point type that satisfies the requested decimal precision and range. 7

7.a **Implementation defined:** What combinations of requested decimal precision and range are supported for floating point types.

Static Semantics

8 The set of values for a floating point type is the (infinite) set of rational numbers. The *machine numbers* of a floating point type are the values of the type that can be represented exactly in every unconstrained variable of the type. The base range (see 3.5) of a floating point type is symmetric around zero, except that it can include some extra negative values in some implementations.

8.a **Implementation Note:** For example, if a 2's complement representation is used for the mantissa rather than a sign-mantissa or 1's complement representation, then there is usually one extra negative machine number.

8.b **To be honest:** If the Signed_Zeros attribute is True, then minus zero could in a sense be considered a value of the type. However, for most purposes, minus zero behaves the same as plus zero.

9 The *base decimal precision* of a floating point type is the number of decimal digits of precision representable in objects of the type. The *safe range* of a floating point type is that part of its base range for which the accuracy corresponding to the base decimal precision is preserved by all predefined operations.

9.a **Implementation Note:** In most cases, the safe range and base range are the same. However, for some hardware, values near the boundaries of the base range might result in excessive inaccuracies or spurious overflows when used with certain predefined operations. For such hardware, the safe range would omit such values.

10/5 {*AI12-0439-1*} A floating_point_definition defines a floating point type whose base decimal precision is no less than the requested decimal precision. If a real_range_specification is given, the safe range of the floating point type (and hence, also its base range) includes at least the values of the simple expressions given in the real_range_specification. If a real_range_specification is not given, the safe (and base) range of the type includes at least the values of the range –10.0**(4*D) .. +10.0**(4*D) where D is the requested decimal precision. [The safe range can include other values as well. The attributes Safe_First and Safe_Last give the actual bounds of the safe range.]

11 A floating_point_definition also defines a first subtype of the type. If a real_range_specification is given, then the subtype is constrained to a range whose bounds are given by a conversion of the values of the simple_expressions of the real_range_specification to the type being defined. Otherwise, the subtype is unconstrained.

11.a.1/1 **To be honest:** The conversion mentioned above is not an *implicit subtype conversion* (which is something that happens at overload resolution, see 4.6), although it happens implicitly. Therefore, the freezing rules are not invoked on the type (which is important so that representation items can be given for the type).

12 There is a predefined, unconstrained, floating point subtype named Float[, declared in the visible part of package Standard].

Dynamic Semantics

13 [The elaboration of a floating_point_definition creates the floating point type and its first subtype.]

Implementation Requirements

14 In an implementation that supports floating point types with 6 or more digits of precision, the requested decimal precision for Float shall be at least 6.

15 If Long_Float is predefined for an implementation, then its requested decimal precision shall be at least 11.

Implementation Permissions

16/5 {*AI12-0444-1*} An implementation is allowed to provide additional predefined floating point types[, declared in the visible part of Standard], whose (unconstrained) first subtypes have names of the form

Short_Float, Long_Float, Short_Short_Float, Long_Long_Float, etc. Different predefined floating point types are allowed to have the same base decimal precision. However, the precision of Float should be no greater than that of Long_Float. Similarly, the precision of Short_Float (if provided) should be no greater than Float. Corresponding recommendations apply to any other predefined floating point types. An implementation may support base decimal precisions for which there is no corresponding named floating point type.

Implementation defined: The predefined floating point types declared in Standard. 16.a

Implementation Advice

An implementation should support Long_Float in addition to Float if the target machine supports 11 or more digits of precision. No other named floating point subtypes are recommended for package Standard. Instead, appropriate named floating point subtypes should be provided in the library package Interfaces (see B.2). 17

Implementation Advice: Long_Float should be declared in Standard if the target supports 11 or more digits of precision. No other named float subtypes should be declared in Standard. 17.a.1/2

Implementation Note: To promote portability, implementations should explicitly declare the floating point (sub)types Float and Long_Float in Standard, and leave other predefined float types anonymous. For implementations that already support Short_Float, etc., upward compatibility argues for keeping such declarations in Standard during the transition period, but perhaps generating a warning on use. A separate package Interfaces in the predefined environment is available for pre-declaring types such as Float_32, IEEE_Float_64, etc. See B.2. 17.a

NOTE If a floating point subtype is unconstrained, then assignments to variables of the subtype involve only Overflow_Checks, never Range_Checks. 18

Examples

Examples of floating point types and subtypes: 19

```
type Coefficient is digits 10 range -1.0 .. 1.0;
```
20

```
type Real is digits 8;
type Mass is digits 7 range 0.0 .. 1.0E35;
```
21

```
subtype Probability is Real range 0.0 .. 1.0;
        -- a subtype with a smaller range
```
22

Inconsistencies With Ada 83

No Range_Checks, only Overflow_Checks, are performed on variables (or parameters) of an unconstrained floating point subtype. This is upward compatible for programs that do not raise Constraint_Error. For those that do raise Constraint_Error, it is possible that the exception will be raised at a later point, or not at all, if extended range floating point registers are used to hold the value of the variable (or parameter). 22.a

Reason: This change was felt to be justified by the possibility of improved performance on machines with extended-range floating point registers. An implementation need not take advantage of this relaxation in the range checking; it can hide completely the use of extended range registers if desired, presumably at some run-time expense. 22.b

Wording Changes from Ada 83

The syntax rules for floating_point_constraint and floating_accuracy_definition are removed. The syntax rules for floating_point_definition and real_range_specification are new. 22.c

A syntax rule for digits_constraint is given in 3.5.9, “Fixed Point Types”. In J.3 we indicate that a digits_constraint may be applied to a floating point subtype_mark as well (to be compatible with Ada 83's floating_point_constraint). 22.d

Discussion of model numbers is postponed to 3.5.8 and G.2. The concept of safe numbers has been replaced by the concept of the safe range of values. The bounds of the safe range are given by T'Safe_First .. T'Safe_Last, rather than -T'Safe_Large .. T'Safe_Large, since on some machines the safe range is not perfectly symmetric. The concept of machine numbers is new, and is relevant to the definition of Succ and Pred for floating point numbers. 22.e

3.5.8 Operations of Floating Point Types

Static Semantics

1 The following attribute is defined for every floating point subtype S:

2/1 S'Digits {*8652/0004*} {*AI95-00203-01*} S'Digits denotes the requested decimal precision for the subtype S. The value of this attribute is of the type *universal_integer*. The requested decimal precision of the base subtype of a floating point type *T* is defined to be the largest value of *d* for which
ceiling(*d* * log(10) / log(T'Machine_Radix)) + *g* <= T'Model_Mantissa
where g is 0 if Machine_Radix is a positive power of 10 and 1 otherwise.

3 NOTE 1 The predefined operations of a floating point type include the assignment operation, qualification, the membership tests, and explicit conversion to and from other numeric types. They also include the relational operators and the following predefined arithmetic operators: the binary and unary adding operators – and +, certain multiplying operators, the unary operator **abs**, and the exponentiation operator.

4 NOTE 2 As for all types, objects of a floating point type have Size and Address attributes (see 13.3). Other attributes of floating point types are defined in A.5.3.

Wording Changes from Ada 95

4.a/2 {*8652/0004*} {*AI95-00203-01*} **Corrigendum:** Corrected the formula for Digits when the Machine_Radix is 10.

3.5.9 Fixed Point Types

1 A fixed point type is either an ordinary fixed point type, or a decimal fixed point type. The error bound of a fixed point type is specified as an absolute value, called the *delta* of the fixed point type.

Syntax

2 fixed_point_definition ::=
ordinary_fixed_point_definition | decimal_fixed_point_definition

3 ordinary_fixed_point_definition ::=
delta *static*_expression real_range_specification

4 decimal_fixed_point_definition ::=
delta *static*_expression **digits** *static*_expression [real_range_specification]

5/4 {*AI12-0152-1*} digits_constraint ::=
digits *static*_simple_expression [range_constraint]

Name Resolution Rules

6 For a type defined by a fixed_point_definition, the *delta* of the type is specified by the value of the expression given after the reserved word **delta**; this expression is expected to be of any real type. For a type defined by a decimal_fixed_point_definition (a *decimal* fixed point type), the number of significant decimal digits for its first subtype (the *digits* of the first subtype) is specified by the expression given after the reserved word **digits**; this expression is expected to be of any integer type.

6.1/4 {*AI12-0159-1*} The simple_expression of a digits_constraint is expected to be of any integer type.

Legality Rules

7 In a fixed_point_definition or digits_constraint, the expressions given after the reserved words **delta** and **digits** shall be static; their values shall be positive.

{*AI95-00100-01*} The set of values of a fixed point type comprise the integral multiples of a number called the *small* of the type. The *machine numbers* of a fixed point type are the values of the type that can be represented exactly in every unconstrained variable of the type. For a type defined by an ordinary_fixed_point_definition (an *ordinary* fixed point type), the *small* may be specified by an attribute_definition_clause (see 13.3); if so specified, it shall be no greater than the *delta* of the type. If not specified, the *small* of an ordinary fixed point type is an implementation-defined power of two less than or equal to the *delta*. 8/2

Implementation defined: The *small* of an ordinary fixed point type. 8.a

For a decimal fixed point type, the *small* equals the *delta*; the *delta* shall be a power of 10. If a real_range_specification is given, both bounds of the range shall be in the range –(10***digits*–1)**delta* .. +(10***digits*–1)**delta*. 9

A fixed_point_definition is illegal if the implementation does not support a fixed point type with the given *small* and specified range or *digits*. 10

Implementation defined: What combinations of *small*, range, and *digits* are supported for fixed point types. 10.a

For a subtype_indication with a digits_constraint, the subtype_mark shall denote a decimal fixed point subtype. 11

To be honest: Or, as an obsolescent feature, a floating point subtype is permitted — see J.3. 11.a

Static Semantics

The base range (see 3.5) of a fixed point type is symmetric around zero, except possibly for an extra negative value in some implementations. 12

An ordinary_fixed_point_definition defines an ordinary fixed point type whose base range includes at least all multiples of *small* that are between the bounds specified in the real_range_specification. The base range of the type does not necessarily include the specified bounds themselves. An ordinary_fixed_-point_definition also defines a constrained first subtype of the type, with each bound of its range given by the closer to zero of: 13

- the value of the conversion to the fixed point type of the corresponding expression of the real_range_specification; 14

  To be honest: The conversion mentioned above is not an *implicit subtype conversion* (which is something that happens at overload resolution, see 4.6), although it happens implicitly. Therefore, the freezing rules are not invoked on the type (which is important so that representation items can be given for the type). 14.a.1/1

- the corresponding bound of the base range. 15

A decimal_fixed_point_definition defines a decimal fixed point type whose base range includes at least the range –(10***digits*–1)**delta* .. +(10***digits*–1)**delta*. A decimal_fixed_point_definition also defines a constrained first subtype of the type. If a real_range_specification is given, the bounds of the first subtype are given by a conversion of the values of the expressions of the real_range_specification. Otherwise, the range of the first subtype is –(10***digits*–1)**delta* .. +(10***digits*–1)**delta*. 16

To be honest: The conversion mentioned above is not an *implicit subtype conversion* (which is something that happens at overload resolution, see 4.6), although it happens implicitly. Therefore, the freezing rules are not invoked on the type (which is important so that representation items can be given for the type). 16.a.1/1

Dynamic Semantics

The elaboration of a fixed_point_definition creates the fixed point type and its first subtype. 17

{*AI12-0152-1*} For a digits_constraint on a decimal fixed point subtype with a given *delta*, if it does not have a range_constraint, then it specifies an implicit range –(10****D*–1)**delta* .. +(10****D*–1)**delta*, where *D* is the value of the simple_expression. A digits_constraint is *compatible* with a decimal fixed 18/4

point subtype if the value of the simple_expression is no greater than the *digits* of the subtype, and if it specifies (explicitly or implicitly) a range that is compatible with the subtype.

18.a **Discussion:** Except for the requirement that the *digits* specified be no greater than the *digits* of the subtype being constrained, a digits_constraint is essentially equivalent to a range_constraint.

18.b Consider the following example:

18.c
```
type D is delta 0.01 digits 7 range -0.00 .. 9999.99;
```

18.d/1 The compatibility rule implies that the digits_constraint "**digits** 6" specifies an implicit range of "–9999.99 .. 9999.99". Thus, "**digits** 6" is not compatible with the constraint of D, but "**digits** 6 range 0.00 .. 9999.99" is compatible.

18.e/2 {*AI95-00114-01*} A value of a scalar type belongs to a constrained subtype of the type if it belongs to the range of the subtype. Attributes like Digits and Delta have no effect on this fundamental rule. So the obsolescent forms of digits_constraints and delta_constraints that are called “accuracy constraints” in RM83 don't really represent constraints on the values of the subtype, but rather primarily affect compatibility of the “constraint” with the subtype being “constrained”. In this sense, they might better be called “subtype assertions” rather than “constraints”.

18.f Note that the digits_constraint on a decimal fixed point subtype is a combination of an assertion about the *digits* of the subtype being further constrained, and a constraint on the range of the subtype being defined, either explicit or implicit.

19/4 {*AI12-0152-1*} The elaboration of a digits_constraint consists of the elaboration of the range_constraint, if any. If a range_constraint is given, a check is made that the bounds of the range are both in the range –(10**D*–1)**delta* .. +(10**D*–1)**delta*, where *D* is the value of the (static) simple_expression given after the reserved word **digits**. If this check fails, Constraint_Error is raised.

Implementation Requirements

20 The implementation shall support at least 24 bits of precision (including the sign bit) for fixed point types.

20.a **Reason:** This is sufficient to represent Standard.Duration with a *small* no more than 50 milliseconds.

Implementation Permissions

21 Implementations are permitted to support only *small*s that are a power of two. In particular, all decimal fixed point type declarations can be disallowed. Note however that conformance with the Information Systems Annex requires support for decimal *small*s, and decimal fixed point type declarations with *digits* up to at least 18.

21.a **Implementation Note:** The accuracy requirements for multiplication, division, and conversion (see G.2.1, “Model of Floating Point Arithmetic”) are such that support for arbitrary *small*s should be practical without undue implementation effort. Therefore, implementations should support fixed point types with arbitrary values for *small* (within reason). One reasonable limitation would be to limit support to fixed point types that can be converted to the most precise floating point type without loss of precision (so that Fixed_IO is implementable in terms of Float_IO).

22/5 NOTE {*AI12-0442-1*} The specified bounds themselves can be outside the base range of an ordinary fixed point type so that the range specification can be given in a natural way, such as:

23
```
type Fraction is delta 2.0**(-15) range -1.0 .. 1.0;
```

24/5 {*AI12-0442-1*} With 2's complement hardware, such a type would typically have a signed 16-bit representation, using 1 bit for the sign and 15 bits for fraction, resulting in a base range of –1.0 .. 1.0–2.0**(–15).

Examples

25 *Examples of fixed point types and subtypes:*

26
```
type Volt is delta 0.125 range 0.0 .. 255.0;
```

27
```
   -- A pure fraction which requires all the available
   -- space in a word can be declared as the type Fraction:
type Fraction is delta System.Fine_Delta range -1.0 .. 1.0;
   -- Fraction'Last = 1.0 – System.Fine_Delta
```

28
```
type Money is delta 0.01 digits 15;  -- decimal fixed point
subtype Salary is Money digits 10;
   -- Money'Last = 10.0**13 – 0.01, Salary'Last = 10.0**8 – 0.01
```

Inconsistencies With Ada 83

In Ada 95, S'Small always equals S'Base'Small, so if an implementation chooses a *small* for a fixed point type smaller than required by the *delta*, the value of S'Small in Ada 95 might not be the same as it was in Ada 83. 28.a

Extensions to Ada 83

{*AI05-0005-1*} Decimal fixed point types are new, though their capabilities are essentially similar to that available in Ada 83 with a fixed point type whose *small* equals its *delta* and both are a power of 10. However, in the Information Systems Annex, additional requirements are placed on the support of decimal fixed point types (e.g. a minimum of 18 digits of precision). 28.b/3

Wording Changes from Ada 83

The syntax rules for fixed_point_constraint and fixed_accuracy_definition are removed. The syntax rule for fixed_point_definition is new. A syntax rule for delta_constraint is included in the Obsolescent features (to be compatible with Ada 83's fixed_point_constraint). 28.c

Wording Changes from Ada 95

{*AI95-00100-01*} Added wording to define the machine numbers of fixed point types; this is needed by the static evaluation rules. 28.d/2

Incompatibilities With Ada 2012

{*AI12-0152-1*} **Corrigendum:** Changed the syntax so that the value following **digits** in a digits_constraint is a simple_expression. This is compatible with one very unlikely exception: if the **digits** expression is a static expression of a modular type using an unparenthesized logical operator (like **and** or **or**). Parenthesizing the expression will make it legal in that case. The change is necessary to eliminate syntax ambguities in derived_type_definitions. 28.e/4

Wording Changes from Ada 2012

{*AI12-0159-1*} **Corrigendum**:Added wording to define the expected type for a digits_constraint. This was missing since Ada 95, but as it is obvious and unchanged from Ada 83, we don't consider it an incompatibility. 28.f/4

3.5.10 Operations of Fixed Point Types

Static Semantics

The following attributes are defined for every fixed point subtype S: 1

S'Small {*8652/0005*} {*AI95-00054-01*} {*AI12-0367-1*} S'Small denotes the *small* of the type of S. The value of this attribute is of the type *universal_real*. Small may be specified for nonderived ordinary fixed point types via an attribute_definition_clause (see 13.3); the expression of such a clause shall be static and positive. 2/5

Aspect Description for Small: Scale factor for a fixed point type. 2.a/3

S'Delta S'Delta denotes the *delta* of the fixed point subtype S. The value of this attribute is of the type *universal_real*. 3

Reason: The *delta* is associated with the *sub*type as opposed to the type, because of the possibility of an (obsolescent) delta_constraint. 3.a

S'Fore S'Fore yields the minimum number of characters needed before the decimal point for the decimal representation of any value of the subtype S, assuming that the representation does not include an exponent, but includes a one-character prefix that is either a minus sign or a space. (This minimum number does not include superfluous zeros or underlines, and is at least 2.) The value of this attribute is of the type *universal_integer*. 4

S'Aft S'Aft yields the number of decimal digits needed after the decimal point to accommodate the *delta* of the subtype S, unless the *delta* of the subtype S is greater than 0.1, in which case the attribute yields the value one. [(S'Aft is the smallest positive integer N for which (10**N)*S'Delta is greater than or equal to one.)] The value of this attribute is of the type *universal_integer*. 5

6 The following additional attributes are defined for every decimal fixed point subtype S:

7 S'Digits S'Digits denotes the *digits* of the decimal fixed point subtype S, which corresponds to the number of decimal digits that are representable in objects of the subtype. The value of this attribute is of the type *universal_integer*. Its value is determined as follows:

8 • For a first subtype or a subtype defined by a subtype_indication with a digits_constraint, the digits is the value of the expression given after the reserved word **digits**;

9/5 • {*AI12-0426-1*} For a subtype defined by a subtype_indication without a digits_constraint, the digits of the subtype is the same as that of the subtype denoted by the subtype_mark in the subtype_indication;

9.a **Implementation Note:** Although a decimal subtype can be both range-constrained and digits-constrained, the digits constraint is intended to control the Size attribute of the subtype. For decimal types, Size can be important because input/output of decimal types is so common.

10 • The digits of a base subtype is the largest integer *D* such that the range –(10**D*–1)**delta* .. +(10**D*–1)**delta* is included in the base range of the type.

11 S'Scale S'Scale denotes the *scale* of the subtype S, defined as the value N such that S'Delta = 10.0**(–N). [The scale indicates the position of the point relative to the rightmost significant digits of values of subtype S.] The value of this attribute is of the type *universal_integer*.

11.a **Ramification:** S'Scale is negative if S'Delta is greater than one. By contrast, S'Aft is always positive.

12 S'Round S'Round denotes a function with the following specification:

13
```
function S'Round(X : universal_real)
  return S'Base
```

14 The function returns the value obtained by rounding X (away from 0, if X is midway between two values of the type of S).

15 NOTE 1 All subtypes of a fixed point type will have the same value for the Delta attribute, in the absence of delta_constraints (see J.3).

16 NOTE 2 S'Scale is not always the same as S'Aft for a decimal subtype; for example, if S'Delta = 1.0 then S'Aft is 1 while S'Scale is 0.

17 NOTE 3 The predefined operations of a fixed point type include the assignment operation, qualification, the membership tests, and explicit conversion to and from other numeric types. They also include the relational operators and the following predefined arithmetic operators: the binary and unary adding operators – and +, multiplying operators, and the unary operator **abs**.

18 NOTE 4 As for all types, objects of a fixed point type have Size and Address attributes (see 13.3). Other attributes of fixed point types are defined in A.5.4.

Wording Changes from Ada 95

18.a/2 {*8652/0005*} {*AI95-00054-01*} **Corrigendum:** Clarified that *small* may be specified only for ordinary fixed point types.

Wording Changes from Ada 2012

18.b/5 {*AI12-0367-1*} **Correction:** Clarified that *small* may be specified only with positive values.

3.6 Array Types

1 An *array* object is a composite object consisting of components which all have the same subtype. The name for a component of an array uses one or more index values belonging to specified discrete types. The value of an array object is a composite value consisting of the values of the components.

Syntax

2 array_type_definition ::=

unconstrained_array_definition | constrained_array_definition

unconstrained_array_definition ::= 3
array(index_subtype_definition {, index_subtype_definition}) **of** component_definition

index_subtype_definition ::= subtype_mark **range** <> 4

constrained_array_definition ::= 5
array (discrete_subtype_definition {, discrete_subtype_definition}) **of** component_definition

discrete_subtype_definition ::= *discrete_*subtype_indication | range 6

{*AI95-00230-01*} {*AI95-00406-01*} component_definition ::= 7/2
[**aliased**] subtype_indication
| [**aliased**] access_definition

Name Resolution Rules

For a discrete_subtype_definition that is a range, the range shall resolve to be of some specific discrete type[; which discrete type shall be determined without using any context other than the bounds of the range itself (plus the preference for *root_integer* — see 8.6).] 8

Legality Rules

Each index_subtype_definition or discrete_subtype_definition in an array_type_definition defines an *index subtype*; its type (the *index type*) shall be discrete. 9

Discussion: An *index* is a discrete quantity used to select along a given dimension of an array. A component is selected by specifying corresponding values for each of the indices. 9.a

The subtype defined by the subtype_indication of a component_definition (the *component subtype*) shall be a definite subtype. 10

Ramification: This applies to all uses of component_definition, including in record_type_definitions and protected_definitions. 10.a

This paragraph was deleted.{*AI95-00363-01*} 11/2

Static Semantics

An array is characterized by the number of indices (the *dimensionality* of the array), the type and position of each index, the lower and upper bounds for each index, and the subtype of the components. The order of the indices is significant. 12

A one-dimensional array has a distinct component for each possible index value. A multidimensional array has a distinct component for each possible sequence of index values that can be formed by selecting one value for each index position (in the given order). The possible values for a given index are all the values between the lower and upper bounds, inclusive; this range of values is called the *index range*. The *bounds* of an array are the bounds of its index ranges. The *length* of a dimension of an array is the number of values of the index range of the dimension (zero for a null range). The *length* of a one-dimensional array is the length of its only dimension. 13

An array_type_definition defines an array type and its first subtype. For each object of this array type, the number of indices, the type and position of each index, and the subtype of the components are as in the type definition[; the values of the lower and upper bounds for each index belong to the corresponding index subtype of its type, except for null arrays (see 3.6.1)]. 14

{*AI12-0444-1*} An unconstrained_array_definition defines an array type with an unconstrained first subtype. Each index_subtype_definition defines the corresponding index subtype to be the subtype denoted by the subtype_mark. [The compound delimiter <> (called a *box*) of an 15/5

index_subtype_definition stands for an undefined range (different objects of the type can have different bounds).]

16 A constrained_array_definition defines an array type with a constrained first subtype. Each discrete_subtype_definition defines the corresponding index subtype, as well as the corresponding index range for the constrained first subtype. The *constraint* of the first subtype consists of the bounds of the index ranges.

16.a/3 **Discussion:** {*AI05-0005-1*} Although there is no nameable unconstrained array subtype in this case, the predefined slicing and concatenation operations can operate on and yield values that do not necessarily belong to the first array subtype. This is also true for Ada 83.

17 The discrete subtype defined by a discrete_subtype_definition is either that defined by the subtype_indication, or a subtype determined by the range as follows:

18 • If the type of the range resolves to *root_integer*, then the discrete_subtype_definition defines a subtype of the predefined type Integer with bounds given by a conversion to Integer of the bounds of the range;

18.a **Reason:** This ensures that indexing over the discrete subtype can be performed with regular Integers, rather than only *universal_integer*s.

18.b **Discussion:** We considered doing this by simply creating a "preference" for Integer when resolving the range. However, this can introduce *Beaujolais* effects when the simple_expressions involve calls on functions visible due to **use** clauses.

19 • Otherwise, the discrete_subtype_definition defines a subtype of the type of the range, with the bounds given by the range.

20 The component_definition of an array_type_definition defines the nominal subtype of the components. If the reserved word **aliased** appears in the component_definition, then each component of the array is aliased (see 3.10).

Dynamic Semantics

21 The elaboration of an array_type_definition creates the array type and its first subtype, and consists of the elaboration of any discrete_subtype_definitions and the component_definition.

22/2 {*8652/0002*} {*AI95-00171-01*} {*AI95-00230-01*} The elaboration of a discrete_subtype_definition that does not contain any per-object expressions creates the discrete subtype, and consists of the elaboration of the subtype_indication or the evaluation of the range. The elaboration of a discrete_subtype_definition that contains one or more per-object expressions is defined in 3.8. The elaboration of a component_definition in an array_type_definition consists of the elaboration of the subtype_indication or access_definition. The elaboration of any discrete_subtype_definitions and the elaboration of the component_definition are performed in an arbitrary order.

Static Semantics

22.1/3 {*AI05-0228-1*} For an array type with a scalar component type, the following language-defined representation aspect may be specified with an aspect_specification (see 13.1.1):

22.2/3 Default_Component_Value
: This aspect shall be specified by a static expression, and that expression shall be explicit, even if the aspect has a boolean type. Default_Component_Value shall be specified only on a full_type_declaration.

22.a/3 **Reason:** The part about requiring an explicit expression is to disallow omitting the value for this aspect, which would otherwise be allowed by the rules of 13.1.1.

22.b/3 This is a representation attribute in order to disallow specifying it on a derived type that has inherited primitive subprograms; that is necessary as the sizes of **out** parameters could be different whether or not a Default_Value is specified (see 6.4.1).

Aspect Description for Default_Component_Value: Default value for the components of an array-of-scalar subtype. 22.c/3

{*AI05-0228-1*} {*AI12-0427-1*} If a derived type inherits a boolean Default_Component_Value aspect, the aspect may be specified to have any value for the derived type. 22.3/5

Reason: This overrides the 13.1.1 rule that says that a boolean aspect with a value True cannot be changed. 22.d/3

Name Resolution Rules

{*AI05-0228-1*} The expected type for the expression specified for the Default_Component_Value aspect is the component type of the array type defined by the full_type_declaration on which it appears. 22.4/3

NOTE 1 All components of an array have the same subtype. In particular, for an array of components that are one-dimensional arrays, this means that all components have the same bounds and hence the same length. 23

NOTE 2 Each elaboration of an array_type_definition creates a distinct array type. A consequence of this is that each object whose object_declaration contains an array_type_definition is of its own unique type. 24

Examples

Examples of type declarations with unconstrained array definitions: 25

```
type Vector     is array(Integer  range <>) of Real;                          26
type Matrix     is array(Integer  range <>, Integer range <>) of Real;
type Bit_Vector is array(Integer  range <>) of Boolean;
type Roman      is array(Positive range <>) of Roman_Digit; -- see 3.5.2
```

Examples of type declarations with constrained array definitions: 27

```
type Table    is array(1 .. 10) of Integer;                                   28
type Schedule is array(Day) of Boolean;
type Line     is array(1 .. Max_Line_Size) of Character;
```

Examples of object declarations with array type definitions: 29

```
{AI95-00433-01} Grid      : array(1 .. 80, 1 .. 100) of Boolean;              30/2
Mix       : array(Color range Red .. Green) of Boolean;
Msg_Table : constant array(Error_Code) of access constant String :=
      (Too_Big => new String'("Result too big"), Too_Small => ...);
Page      : array(Positive range <>) of Line := -- an array of arrays
  (1 | 50  => Line'(1 | Line'Last => '+', others => '-'), -- see 4.3.3
   2 .. 49 => Line'(1 | Line'Last => '|', others => ' '));
    -- Page is constrained by its initial value to (1..50)
```

Extensions to Ada 83

The syntax rule for component_definition is modified to allow the reserved word **aliased**. 30.a

The syntax rules for unconstrained_array_definition and constrained_array_definition are modified to use component_definition (instead of *component*_subtype_indication). The effect of this change is to allow the reserved word **aliased** before the component subtype_indication. 30.b

A range in a discrete_subtype_definition may use arbitrary universal expressions for each bound (e.g. –1 .. 3+5), rather than strictly "implicitly convertible" operands. The subtype defined will still be a subtype of Integer. 30.c

Wording Changes from Ada 83

We introduce a new syntactic category, discrete_subtype_definition, as distinct from discrete_range. These two constructs have the same syntax, but their semantics are quite different (one defines a subtype, with a preference for Integer subtypes, while the other just selects a subrange of an existing subtype). We use this new syntactic category in **for** loops and entry families. 30.d

The syntax for index_constraint and discrete_range have been moved to their own subclause, since they are no longer used here. 30.e

The syntax rule for component_definition (formerly component_subtype_definition) is moved here from RM83-3.7. 30.f

Extensions to Ada 95

{*AI95-00230-01*} {*AI95-00406-01*} Array components can have an anonymous access type. 30.g/2

30.h/2 {*AI95-00363-01*} The prohibition against unconstrained discriminated aliased components has been lifted. It has been replaced by a prohibition against the actual troublemakers: general access discriminant constraints (see 3.7.1).

Wording Changes from Ada 95

30.i/2 {*8652/0002*} {*AI95-00171-01*} **Corrigendum:** Added wording to allow the elaboration of per-object constraints for constrained arrays.

Extensions to Ada 2005

30.j/3 {*AI05-0228-1*} The new aspect Default_Component_Value allows defining implicit initial values (see 3.3.1) for arrays of scalar types.

3.6.1 Index Constraints and Discrete Ranges

1 An index_constraint determines the range of possible values for every index of an array subtype, and thereby the corresponding array bounds.

Syntax

2 index_constraint ::= (discrete_range {, discrete_range})

3 discrete_range ::= *discrete*_subtype_indication | range

Name Resolution Rules

4 The type of a discrete_range is the type of the subtype defined by the subtype_indication, or the type of the range. For an index_constraint, each discrete_range shall resolve to be of the type of the corresponding index.

4.a **Discussion:** In Ada 95, index_constraints only appear in a subtype_indication; they no longer appear in constrained_array_definitions.

Legality Rules

5 An index_constraint shall appear only in a subtype_indication whose subtype_mark denotes either an unconstrained array subtype, or an unconstrained access subtype whose designated subtype is an unconstrained array subtype; in either case, the index_constraint shall provide a discrete_range for each index of the array type.

Static Semantics

6 A discrete_range defines a range whose bounds are given by the range, or by the range of the subtype defined by the subtype_indication.

Dynamic Semantics

7 An index_constraint is *compatible* with an unconstrained array subtype if and only if the index range defined by each discrete_range is compatible (see 3.5) with the corresponding index subtype. If any of the discrete_ranges defines a null range, any array thus constrained is a *null array*, having no components. An array value *satisfies* an index_constraint if at each index position the array value and the index_constraint have the same index bounds.

7.a **Ramification:** There is no need to define compatibility with a constrained array subtype, because one is not allowed to constrain it again.

8 The elaboration of an index_constraint consists of the evaluation of the discrete_range(s), in an arbitrary order. The evaluation of a discrete_range consists of the elaboration of the subtype_indication or the evaluation of the range.

9 NOTE 1 The elaboration of a subtype_indication consisting of a subtype_mark followed by an index_constraint checks the compatibility of the index_constraint with the subtype_mark (see 3.2.2).

NOTE 2 Even if an array value does not satisfy the index constraint of an array subtype, Constraint_Error is not raised on conversion to the array subtype, so long as the length of each dimension of the array value and the array subtype match. See 4.6. 10

Examples

Examples of array declarations including an index constraint: 11

```
{AI12-0442-1} Board     : Matrix(1 .. 8,  1 .. 8);  -- see 3.6                12/5
Rectangle : Matrix(1 .. 20, 1 .. 30);
Inverse   : Matrix(1 .. N,  1 .. N);  -- N can be nonstatic

{AI125-0430-1} Filter    : Bit_Vector(0 .. 31);       -- see 3.6               13/5
```

Example of array declaration with a constrained array subtype: 14

```
My_Schedule : Schedule;  -- all arrays of type Schedule have the same bounds   15
```

Example of record type with a component that is an array: 16

```
type Var_Line(Length : Natural) is                                             17
   record
      Image : String(1 .. Length);
   end record;

Null_Line : Var_Line(0);  -- Null_Line.Image is a null array                   18
```

Extensions to Ada 83

We allow the declaration of a variable with a nominally unconstrained array subtype, so long as it has an initialization expression to determine its bounds. 18.a

Wording Changes from Ada 83

We have moved the syntax for index_constraint and discrete_range here since they are no longer used in constrained_array_definitions. We therefore also no longer have to describe the (special) semantics of index_constraints and discrete_ranges that appear in constrained_array_definitions. 18.b

The rules given in RM83-3.6.1(5,7-10), which define the bounds of an array object, are redundant with rules given elsewhere, and so are not repeated here. RM83-3.6.1(6), which requires that the (nominal) subtype of an array variable be constrained, no longer applies, so long as the variable is explicitly initialized. 18.c

3.6.2 Operations of Array Types

Legality Rules

[The argument N used in the attribute_designators for the N-th dimension of an array shall be a static expression of some integer type.] The value of N shall be positive (nonzero) and no greater than the dimensionality of the array. 1

Static Semantics

{*8652/0006*} {*AI95-00030-01*} The following attributes are defined for a prefix A that is of an array type [(after any implicit dereference)], or denotes a constrained array subtype: 2/1

Ramification: These attributes are not defined if A is a subtype-mark for an access-to-array subtype. They are defined (by implicit dereference) for access-to-array values. 2.a

A'First A'First denotes the lower bound of the first index range; its type is the corresponding index type. 3

A'First(N) A'First(N) denotes the lower bound of the N-th index range; its type is the corresponding index type. 4

A'Last A'Last denotes the upper bound of the first index range; its type is the corresponding index type. 5

6 A'Last(N) A'Last(N) denotes the upper bound of the N-th index range; its type is the corresponding index type.

7 A'Range A'Range is equivalent to the range A'First .. A'Last, except that the prefix A is only evaluated once.

8 A'Range(N)
A'Range(N) is equivalent to the range A'First(N) .. A'Last(N), except that the prefix A is only evaluated once.

9 A'Length A'Length denotes the number of values of the first index range (zero for a null range); its type is *universal_integer*.

10 A'Length(N)
A'Length(N) denotes the number of values of the N-th index range (zero for a null range); its type is *universal_integer*.

Implementation Advice

11/3 {*AI05-0229-1*} An implementation should normally represent multidimensional arrays in row-major order, consistent with the notation used for multidimensional array aggregates (see 4.3.3). However, if convention Fortran is specified for a multidimensional array type, then column-major order should be used instead (see B.5, “Interfacing with Fortran”).

11.a/2 **Implementation Advice:** Multidimensional arrays should be represented in row-major order, unless the array has convention Fortran.

12 NOTE 1 The attribute_references A'First and A'First(1) denote the same value. A similar relation exists for the attribute_references A'Last, A'Range, and A'Length. The following relation is satisfied (except for a null array) by the above attributes if the index type is an integer type:

13
```
A'Length(N) = A'Last(N) - A'First(N) + 1
```

14 NOTE 2 An array type is limited if its component type is limited (see 7.5).

15 NOTE 3 The predefined operations of an array type include the membership tests, qualification, and explicit conversion. If the array type is not limited, they also include assignment and the predefined equality operators. For a one-dimensional array type, they include the predefined concatenation operators (if nonlimited) and, if the component type is discrete, the predefined relational operators; if the component type is boolean, the predefined logical operators are also included.

16/2 NOTE 4 {*AI95-00287-01*} A component of an array can be named with an indexed_component. A value of an array type can be specified with an array_aggregate. For a one-dimensional array type, a slice of the array can be named; also, string literals are defined if the component type is a character type.

Examples

17/5 {*AI12-0449-1*} *Examples (using arrays declared in the examples of 3.6.1):*

18
```
-- Filter'First      =  0   Filter'Last       = 31   Filter'Length = 32
-- Rectangle'Last(1) = 20   Rectangle'Last(2) = 30
```

3.6.3 String Types

Static Semantics

1 A one-dimensional array type whose component type is a character type is called a *string type*.

2/2 {*AI95-00285-01*} [There are three predefined string types, String, Wide_String, and Wide_Wide_String, each indexed by values of the predefined subtype Positive; these are declared in the visible part of package Standard:]

3
```
[subtype Positive is Integer range 1 .. Integer'Last;
```

4/2
```
{AI95-00285-01} type String is array(Positive range <>) of Character;
type Wide_String is array(Positive range <>) of Wide_Character;
type Wide_Wide_String is array(Positive range <>) of Wide_Wide_Character;
]
```

NOTE String literals (see 2.6 and 4.2) are defined for all string types. The concatenation operator & is predefined for string types, as for all nonlimited one-dimensional array types. The ordering operators <, <=, >, and >= are predefined for string types, as for all one-dimensional discrete array types; these ordering operators correspond to lexicographic order (see 4.5.2). 5

Examples

Examples of string objects: 6

```
Stars      : String(1 .. 120) := (1 .. 120 => '*' );                    7
Question   : constant String  := "How many characters?";
                      -- Question'First = 1, Question'Last = 20
                      -- Question'Length = 20 (the number of characters)
Ask_Twice  : String  := Question & Question; -- constrained to (1..40)  8
Ninety_Six : constant Roman   := "XCVI";     -- see 3.5.2 and 3.6
```

Inconsistencies With Ada 83

The declaration of Wide_String in Standard hides a use-visible declaration with the same defining_identifier. In rare cases, this might result in an inconsistency between Ada 83 and Ada 95. 8.a

Incompatibilities With Ada 83

Because both String and Wide_String are always directly visible, an expression like 8.b

```
"a" < "bc"                                                             8.c
```

is now ambiguous, whereas in Ada 83 both string literals could be resolved to type String. 8.d

Extensions to Ada 83

The type Wide_String is new (though it was approved by ARG for Ada 83 compilers as well). 8.e

Wording Changes from Ada 83

We define the term *string type* as a natural analogy to the term *character type*. 8.f

Inconsistencies With Ada 95

{*AI95-00285-01*} The declaration of Wide_Wide_String in Standard hides a use-visible declaration with the same defining_identifier. In the (very) unlikely event that an Ada 95 program had depended on such a use-visible declaration, and the program remains legal after the substitution of Standard.Wide_Wide_String, the meaning of the program will be different. 8.g/2

Extensions to Ada 95

{*AI95-00285-01*} The type Wide_Wide_String is new. 8.h/2

3.7 Discriminants

{*AI95-00251-01*} {*AI95-00326-01*} [A composite type (other than an array or interface type) can have discriminants, which parameterize the type. A known_discriminant_part specifies the discriminants of a composite type. A discriminant of an object is a component of the object, and is either of a discrete type or an access type. An unknown_discriminant_part in the declaration of a view of a type specifies that the discriminants of the type are unknown for the given view; all subtypes of such a view are indefinite subtypes.] 1/2

Term entry: discriminant — parameter for a composite type, which can control, for example, the bounds of a component that is an array 1.a/5
Note: A discriminant for a task type can be used to pass data to a task of the type upon its creation.

Discussion: {*AI95-00114-01*} A view of a type, and all subtypes of the view, have *unknown discriminants* when the number or names of the discriminants, if any, are unknown at the point of the type declaration for the view. A discriminant_part of (<>) is used to indicate unknown discriminants. 1.b/2

Language Design Principles

1.c/2 {*AI95-00402-01*} When an access discriminant is initialized at the time of object creation with an allocator of an anonymous type, the allocated object and the object with the discriminant are tied together for their lifetime. They should be allocated out of the same storage pool, and then at the end of the lifetime of the enclosing object, finalized and reclaimed together. In this case, the allocated object is called a coextension (see 3.10.2).

1.d/2 **Discussion:** The above principle when applied to a nonlimited type implies that such an object may be copied only to a shorter-lived object, because attempting to assign it to a longer-lived object would fail because the access discriminants would not match. In a copy, the lifetime connection between the enclosing object and the allocated object does not exist. The allocated object is tied in the above sense only to the original object. Other copies have only secondary references to it.

1.e/2 Note that when an allocator appears as a constraint on an access discriminant in a subtype_indication that is elaborated independently from object creation, no such connection exists. For example, if a named constrained subtype is declared via "**subtype** Constr **is** Rec(Acc_Discrim => **new** T);" or if such an allocator appears in the subtype_indication for a component, the allocator is evaluated when the subtype_indication is elaborated, and hence its lifetime is typically longer than the objects or components that will later be subject to the constraint. In these cases, the allocated object should not be reclaimed until the subtype_indication goes out of scope.

Syntax

2 discriminant_part ::= unknown_discriminant_part | known_discriminant_part

3 unknown_discriminant_part ::= (<>)

4 known_discriminant_part ::=
  (discriminant_specification {; discriminant_specification})

5/5 {*AI95-00231-01*} {*AI12-0398-1*} discriminant_specification ::=
  defining_identifier_list : [null_exclusion] subtype_mark [:= default_expression]
    [aspect_specification]
 | defining_identifier_list : access_definition [:= default_expression]
    [aspect_specification]

5.a/5 **Discussion:** {*AI12-0398-1*} Only implementation-defined aspects are allowed on discriminants in Ada 2022. Implementers are cautioned that any aspect allowed on a discriminant will need conformance rules. If, for instance, an aspect changed the representation of a discriminant, rules would be needed to ensure that the representation is the same for all views of the type (partial and full).

6 default_expression ::= expression

Name Resolution Rules

7 The expected type for the default_expression of a discriminant_specification is that of the corresponding discriminant.

Legality Rules

8/2 {*8652/0007*} {*AI95-00098-01*} {*AI95-00251-01*} A discriminant_part is only permitted in a declaration for a composite type that is not an array or interface type [(this includes generic formal types)]. A type declared with a known_discriminant_part is called a *discriminated* type, as is a type that inherits (known) discriminants.

8.a **Implementation Note:** Discriminants on array types were considered, but were omitted to ease (existing) implementations.

8.b **Discussion:** Note that the above definition for “discriminated type” does not include types declared with an unknown_discriminant_part. This seems consistent with Ada 83, where such types (in a generic formal part) would not be considered discriminated types. Furthermore, the full type for a type with unknown discriminants need not even be composite, much less have any discriminants.

8.b.1/1 {*8652/0007*} {*AI95-00098-01*} On the other hand, unknown_discriminant_parts cannot be applied to type declarations that cannot have a known_discriminant_part. There is no point in having unknown discriminants on a type that can never have discriminants (for instance, a formal modular type), even when these are allowed syntactically.

{*AI95-00231-01*} {*AI95-00254-01*} The subtype of a discriminant may be defined by an optional null_exclusion and a subtype_mark, in which case the subtype_mark shall denote a discrete or access subtype, or it may be defined by an access_definition. A discriminant that is defined by an access_definition is called an *access discriminant* and is of an anonymous access type. 9/2

This paragraph was deleted.{*AI95-00230-01*} 9.a/2

Reason: Note that discriminants of a named access type are not considered "access discriminants". Similarly, "access parameter" only refers to a formal parameter defined by an access_definition. 9.b

{*AI95-00402-01*} {*AI05-0214-1*} Default_expressions shall be provided either for all or for none of the discriminants of a known_discriminant_part. No default_expressions are permitted in a known_discriminant_part in a declaration of a nonlimited tagged type [or a generic formal type]. 9.1/3

Reason: The all-or-none rule is related to the rule that a discriminant constraint shall specify values for all discriminants. One could imagine a different rule that allowed a constraint to specify only some of the discriminants, with the others provided by default. Having defaults for discriminants has a special significance — it allows objects of the type to be unconstrained, with the discriminants alterable as part of assigning to the object. 9.c/2

{*AI05-0214-1*} Defaults for discriminants of tagged types are disallowed so that every object of a nonlimited tagged type is constrained, either by an explicit constraint, or by its initial discriminant values. This substantially simplifies the semantic rules and the implementation of inherited dispatching operations. We don't need this rule for limited tagged types, as the discriminants of such objects cannot be changed after the object is created in any case — no full-object assignment is supported, and that is required to change discriminant values. For generic formal types, the restriction simplifies the type matching rules. If one simply wants a "default" value for the discriminants, a constrained subtype can be declared for future use. 9.d/3

{*AI95-00230-01*} {*AI95-00402-01*} {*AI95-00419-01*} {*AI05-0063-1*} A discriminant_specification for an access discriminant may have a default_expression only in the declaration for an immutably limited type (see 7.5). In addition to the places where Legality Rules normally apply (see 12.3), this rule applies also in the private part of an instance of a generic unit. 10/3

Discussion: This rule implies that a type can have a default for an access discriminant if the type is limited, but not if the only reason it's limited is because of a limited component. Compare the definition of limited type and immutably limited type in 7.5. 10.a/3

Ramification: A (nonformal) limited private type can always have a default for an access discriminant, because having the default itself makes the type immutably limited. Such a private type must necessarily have a full type with the same access discriminant with a default, and thus the full type will always be immutably limited (if legal). 10.b/3

Reason: {*AI95-00230-01*} We considered the following rules for access discriminants: 10.c/2

- If a type has an access discriminant, this automatically makes it limited, just like having a limited component automatically makes a type limited. This was rejected because it decreases program readability, and because it seemed error prone (two bugs in a previous version of the RM9X were attributable to this rule). 10.d
- A type with an access discriminant shall be limited. This is equivalent to the rule we actually chose for Ada 95, except that it allows a type to have an access discriminant if it is limited just because of a limited component. For example, any record containing a task would be allowed to have an access discriminant, whereas the actual rule requires "**limited record**". This rule was also rejected due to readability concerns, and because would interact badly with the rules for limited types that "become nonlimited". 10.e/2
- {*AI05-0063-1*} A type may have an access discriminant if it is an immutably limited type. This was the rule chosen for Ada 95. 10.e.1/3
- Any type may have an access discriminant. For nonlimited type, there is no special accessibility for access discriminants; they're the same as any other anonymous access component. For a limited type, they have the special accessibility of Ada 95. However, this doesn't work because a limited partial view can have a nonlimited full view -- giving the two views different accessibility. 10.f/2
- {*AI05-0063-1*} Any type may have an access discriminant, as above. However, special accessibility rules only apply to types that are immutably limited (task, protected, and explicitly limited records). However, this breaks privacy; worse, Legality Rules depend on the definition of accessibility. 10.f.1/3
- {*AI05-0063-1*} Any type may have an access discriminant, as above. Limited types have special accessibility, while nonlimited types have normal accessibility. However, a limited partial view with an access discriminant can only be completed by an immutably limited type. That prevents accessibility from changing. A runtime accessibility check is required on generic formal types with access discriminants. However, changing between limited and nonlimited types would have far-reaching consequences for access discriminants — which is uncomfortable. 10.f.2/3

10.g/2 - Any type may have an access discriminant. All types have special accessibility. This was considered early during the Ada 9X process, but was dropped for “unpleasant complexities”, which unfortunately aren't recorded. It does seem that an accessibility check would be needed on assignment of such a type, to avoid copying an object with a discriminant pointing to a local object into a more global object (and thus creating a dangling pointer).

10.h/2 - Any type may have an access discriminant, but access discriminants cannot have defaults. All types have special accessibility. This gets rid of the problems on assignment (you couldn't change such a discriminant), but it would be horribly incompatible with Ada 95.

10.h.1/3 - {*AI05-0063-1*} Any type may have an access discriminant, but access discriminants may have defaults only if they are of an immutably limited type. This is the rule chosen for Ada 2005, as it is not incompatible, and it doesn't require weird accessibility checks.

11/2 *This paragraph was deleted.*{*AI95-00402-01*}

12 For a type defined by a derived_type_definition, if a known_discriminant_part is provided in its declaration, then:

13 - The parent subtype shall be constrained;

14 - If the parent type is not a tagged type, then each discriminant of the derived type shall be used in the constraint defining the parent subtype;

14.a **Implementation Note:** This ensures that the new discriminant can share storage with an existing discriminant.

15 - If a discriminant is used in the constraint defining the parent subtype, the subtype of the discriminant shall be statically compatible (see 4.9.1) with the subtype of the corresponding parent discriminant.

15.a **Reason:** This ensures that on conversion (or extension via an extension aggregate) to a distantly related type, if the discriminants satisfy the target type's requirements they satisfy all the intermediate types' requirements as well.

15.b **Ramification:** There is no requirement that the new discriminant have the same (or any) default_expression as the parent's discriminant.

16/3 *This paragraph was deleted.*{*AI05-0102-1*}

16.a/3 *This paragraph was deleted.*

Static Semantics

17 A discriminant_specification declares a discriminant; the subtype_mark denotes its subtype unless it is an access discriminant, in which case the discriminant's subtype is the anonymous access-to-variable subtype defined by the access_definition.

18 [For a type defined by a derived_type_definition, each discriminant of the parent type is either inherited, constrained to equal some new discriminant of the derived type, or constrained to the value of an expression.] When inherited or constrained to equal some new discriminant, the parent discriminant and the discriminant of the derived type are said to *correspond*. Two discriminants also correspond if there is some common discriminant to which they both correspond. A discriminant corresponds to itself as well. If a discriminant of a parent type is constrained to a specific value by a derived_type_definition, then that discriminant is said to be *specified* by that derived_type_definition.

18.a **Ramification:** The correspondence relationship is transitive, symmetric, and reflexive. That is, if A corresponds to B, and B corresponds to C, then A, B, and C each corresponds to A, B, and C in all combinations.

19 A constraint that appears within the definition of a discriminated type *depends on a discriminant* of the type if it names the discriminant as a bound or discriminant value. A component_definition depends on a discriminant if its constraint depends on the discriminant, or on a discriminant that corresponds to it.

19.a **Ramification:** A constraint in a task_body is not considered to *depend* on a discriminant of the task type, even if it names it. It is only the constraints in the type definition itself that are considered dependents. Similarly for protected types.

A component *depends on a discriminant* if: 20

- Its component_definition depends on the discriminant; or 21

  Ramification: A component does *not* depend on a discriminant just because its default_expression refers to the discriminant. 21.a

- It is declared in a variant_part that is governed by the discriminant; or 22

- It is a component inherited as part of a derived_type_definition, and the constraint of the *parent*_subtype_indication depends on the discriminant; or 23

  Reason: When the parent subtype depends on a discriminant, the parent part of the derived type is treated like a discriminant-dependent component. 23.a

  Ramification: Because of this rule, we don't really need to worry about “corresponding” discriminants, since all the inherited components will be discriminant-dependent if there is a new known_discriminant_part whose discriminants are used to constrain the old discriminants. 23.b

- It is a subcomponent of a component that depends on the discriminant. 24

  Reason: The concept of discriminant-dependent (sub)components is primarily used in various rules that disallow renaming or 'Access, or specify that certain discriminant-changing assignments are erroneous. The goal is to allow implementations to move around or change the size of discriminant-dependent subcomponents upon a discriminant-changing assignment to an enclosing object. The above definition specifies that all subcomponents of a discriminant-dependent component or parent part are themselves discriminant-dependent, even though their presence or size does not in fact depend on a discriminant. This is because it is likely that they will move in a discriminant-changing assignment if they are a component of one of several discriminant-dependent parts of the same record. 24.a

Each value of a discriminated type includes a value for each component of the type that does not depend on a discriminant[; this includes the discriminants themselves]. The values of discriminants determine which other component values are present in the value of the discriminated type. 25

To be honest: Which values are present might depend on discriminants of some ancestor type that are constrained in an intervening derived_type_definition. That's why we say "values of discriminants" instead of "values of *the* discriminants" — a subtle point. 25.a

A type declared with a known_discriminant_part is said to have *known discriminants*; its first subtype is unconstrained. A type declared with an unknown_discriminant_part is said to have *unknown discriminants*. A type declared without a discriminant_part has no discriminants, unless it is a derived type; if derived, such a type has the same sort of discriminants (known, unknown, or none) as its parent (or ancestor) type. A tagged class-wide type also has unknown discriminants. [Any subtype of a type with unknown discriminants is an unconstrained and indefinite subtype (see 3.2 and 3.3).] 26

Discussion: {*AI95-00114-01*} An unknown_discriminant_part “(<>)” is only permitted in the declaration of a (generic or nongeneric) private type, private extension, incomplete type, or formal derived type. Hence, only such types, descendants thereof, and class-wide types can have unknown discriminants. An unknown_discriminant_part is used to indicate that the corresponding actual or full type might have discriminants without defaults, or be an unconstrained array subtype. Tagged class-wide types are also considered to have unknown discriminants because discriminants can be added by type extensions, so the total number of discriminants of any given value of a tagged class-wide type is not known at compile time. 26.a/2

{*AI95-00287-01*} A subtype with unknown discriminants is indefinite, and hence an object of such a subtype needs explicit initialization. A limited private type with unknown discriminants is “extremely” limited; objects of such a type can be initialized only by subprograms (either procedures with a parameter of the type, or a function returning the type) declared in the package. Subprograms declared elsewhere can operate on and even return the type, but they can only initialize the object by calling (ultimately) a subprogram in the package declaring the type. Such a type is useful for keeping complete control over object creation within the package declaring the type. 26.b/2

A partial view of a type might have unknown discriminants, while the full view of the same type might have known, unknown, or no discriminants. 26.c

Dynamic Semantics

{*AI95-00230-01*} {*AI95-00416-01*} For an access discriminant, its access_definition is elaborated when the value of the access discriminant is defined: by evaluation of its default_expression, by elaboration of a discriminant_constraint, or by an assignment that initializes the enclosing object. 27/2

27.a/2 **Ramification:** {*AI95-00231-01*} {*AI95-00416-01*} The conversion of the expression defining the access discriminant to the anonymous access type raises Program_Error for an object created by an allocator of an access type T, if the initial value is an access parameter that designates a view whose accessibility level is deeper than that of T.

28 NOTE 1 If a discriminated type has default_expressions for its discriminants, then unconstrained variables of the type are permitted, and the values of the discriminants can be changed by an assignment to such a variable. If defaults are not provided for the discriminants, then all variables of the type are constrained, either by explicit constraint or by their initial value; the values of the discriminants of such a variable cannot be changed after initialization.

28.a **Discussion:** This connection between discriminant defaults and unconstrained variables can be a source of confusion. For Ada 95, we considered various ways to break the connection between defaults and unconstrainedness, but ultimately gave up for lack of a sufficiently simple and intuitive alternative.

28.b An unconstrained discriminated subtype with defaults is called a *mutable* subtype, and a variable of such a subtype is called a mutable variable, because the discriminants of such a variable can change. There are no mutable arrays (that is, the bounds of an array object can never change), because there is no way in the language to define default values for the bounds. Similarly, there are no mutable class-wide subtypes, because there is no way to define the default tag, and defaults for discriminants are not allowed in the tagged case. Mutable tags would also require a way for the maximum possible size of such a class-wide subtype to be known. (In some implementations, all mutable variables are allocated with the maximum possible size. This approach is appropriate for real-time applications where implicit use of the heap is inappropriate.)

29 NOTE 2 The default_expression for a discriminant of a type is evaluated when an object of an unconstrained subtype of the type is created.

30 NOTE 3 Assignment to a discriminant of an object (after its initialization) is not allowed, since the name of a discriminant is a constant; neither assignment_statements nor assignments inherent in passing as an **in out** or **out** parameter are allowed. Note however that the value of a discriminant can be changed by assigning to the enclosing object, presuming it is an unconstrained variable.

30.a/2 **Discussion:** {*AI95-00114-01*} An unknown_discriminant_part is permitted only in the declaration of a private type (including generic formal private), private extension, incomplete type, or generic formal derived type. These are the things that will have a corresponding completion or generic actual, which will either define the discriminants, or say there are none. The (<>) indicates that the actual/full subtype might be an indefinite subtype. An unknown_discriminant_part is not permitted in a normal untagged derived type declaration, because there is no separate full type declaration for such a type. Note that (<>) allows unconstrained array bounds; those are somewhat like undefaulted discriminants.

30.b For a derived type, either the discriminants are inherited as is, or completely respecified in a new discriminant_part. In this latter case, each discriminant of the parent type shall be constrained, either to a specific value, or to equal one of the new discriminants. Constraining a parent type's discriminant to equal one of the new discriminants is like a renaming of the discriminant, except that the subtype of the new discriminant can be more restrictive than that of the parent's one. In any case, the new discriminant can share storage with the parent's discriminant.

31 NOTE 4 A discriminant that is of a named access type is not called an access discriminant; that term is used only for discriminants defined by an access_definition.

Examples

32 *Examples of discriminated types:*

33
```
type Buffer(Size : Buffer_Size := 100)  is          -- see 3.5.4
   record
      Pos   : Buffer_Size := 0;
      Value : String(1 .. Size);
   end record;
```

34
```
type Matrix_Rec(Rows, Columns : Integer) is
   record
      Mat : Matrix(1 .. Rows, 1 .. Columns);        -- see 3.6
   end record;
```

35
```
type Square(Side : Integer) is new
   Matrix_Rec(Rows => Side, Columns => Side);
```

36
```
type Double_Square(Number : Integer) is
   record
      Left  : Square(Number);
      Right : Square(Number);
   end record;
```

```
{AI95-00433-01} {AI05-0229-1} task type Worker(Prio : System.Priority; Buf : access Buffer)
   with Priority => Prio is -- see D.1
   -- discriminants used to parameterize the task type (see 9.1)
   entry Fill;
   entry Drain;
end Worker;
```

37/3

Extensions to Ada 83

The syntax for a discriminant_specification is modified to allow an *access discriminant*, with a type specified by an access_definition (see 3.10). 37.a

{*AI95-00251-01*} Discriminants are allowed on all composite types other than array and interface types. 37.b/2

Discriminants may be of an access type. 37.c

Wording Changes from Ada 83

Discriminant_parts are not elaborated, though an access_definition is elaborated when the discriminant is initialized. 37.d

Extensions to Ada 95

{*AI95-00230-01*} {*AI95-00402-01*} {*AI95-00416-01*} Access discriminants (anonymous access types used as a discriminant) can be used on any type allowing discriminants. Defaults aren't allowed on discriminants of nonlimited types, however, so that accessibility problems don't happen on assignment. 37.e/2

{*AI95-00231-01*} null_exclusion can be used in the declaration of a discriminant. 37.f/2

Wording Changes from Ada 95

{*8652/0007*} {*AI95-00098-01*} **Corrigendum:** The wording was clarified so that types that cannot have discriminants cannot have an unknown_discriminant_part. 37.g/2

{*AI95-00251-01*} Added wording to prevent interfaces from having discriminants. We don't want interfaces to have any components. 37.h/2

{*AI95-00254-01*} Removed wording which implied or required an access discriminant to have an access-to-object type (anonymous access types can now be access-to-subprogram types as well). 37.i/2

{*AI95-00326-01*} {*AI05-0299-1*} Fixed the wording of the introduction to this subclause to reflect that both incomplete and partial views can have unknown discriminants. That was always true, but for some reason this wording specified partial views. 37.j/3

{*AI95-00419-01*} Changed the wording to use the new term “explicitly limited record”, which makes the intent much clearer (and eliminates confusion with derived types that happen to contain the reserved word **limited**). 37.k/2

Incompatibilities With Ada 2005

{*AI05-0063-1*} **Correction:** Changed the rules for when access discriminants can have defaults to depend on the new definition for immutably limited types; this will help ensure that unusual corner cases are properly handled. Note that the Ada 2005 rule was unintentionally incompatible with the Ada 95 rule (as enforced by the ACATS); this change brings it back into alignment with actual practice. So there should be no practical incompatibility. 37.l/3

Extensions to Ada 2005

{*AI05-0214-1*} A limited tagged type may now have defaults for its discriminants. 37.m/3

Wording Changes from Ada 2005

{*AI05-0102-1*} **Correction:** Moved implicit conversion Legality Rule to 8.6. 37.n/3

Extensions to Ada 2012

{*AI12-0398-1*} Discriminants now can have an aspect_specification, allowing the specification of (implementation-defined) aspects for individual discriminants. 37.o/5

3.7.1 Discriminant Constraints

A discriminant_constraint specifies the values of the discriminants for a given discriminated type. 1

Language Design Principles

1.a/3 {*AI05-0299-1*} The rules in this subclause are intentionally parallel to those given in 4.3.1, “Record Aggregates”.

Syntax

2 discriminant_constraint ::=
(discriminant_association {, discriminant_association})

3/5 {*AI12-0212-1*} discriminant_association ::=
[*discriminant*_selector_name {'|' *discriminant*_selector_name} =>] expression

4 A discriminant_association is said to be *named* if it has one or more *discriminant*_selector_names; it is otherwise said to be *positional*. In a discriminant_constraint, any positional associations shall precede any named associations.

Name Resolution Rules

5 Each selector_name of a named discriminant_association shall resolve to denote a discriminant of the subtype being constrained; the discriminants so named are the *associated discriminants* of the named association. For a positional association, the *associated discriminant* is the one whose discriminant_specification occurred in the corresponding position in the known_discriminant_part that defined the discriminants of the subtype being constrained.

6 The expected type for the expression in a discriminant_association is that of the associated discriminant(s).

Legality Rules

7/3 {*8652/0008*} {*AI95-00168-01*} {*AI95-00363-01*} {*AI05-0041-1*} A discriminant_constraint is only allowed in a subtype_indication whose subtype_mark denotes either an unconstrained discriminated subtype, or an unconstrained access subtype whose designated subtype is an unconstrained discriminated subtype. However, in the case of an access subtype, a discriminant_constraint is legal only if any dereference of a value of the access type is known to be constrained (see 3.3). In addition to the places where Legality Rules normally apply (see 12.3), these rules apply also in the private part of an instance of a generic unit.

7.a.1/2 *This paragraph was deleted.*{*8652/0008*} {*AI95-00168-01*} {*AI95-00363-01*}

7.a/2 **Reason:** {*AI95-00363-01*} The second rule is necessary to prevent objects from changing so that they no longer match their constraint. In Ada 95, we attempted to prevent this by banning every case where an aliased object could be unconstrained or be changed by an enclosing assignment. New ways to cause this problem were being discovered frequently, meaning that new rules had to be dreamed up to cover them. Meanwhile, aliased objects and components were getting more and more limited. In Ada 2005, we sweep away all of that cruft and replace it by a simple rule “thou shalt not create an access subtype that can point to an item whose discriminants can be changed by assignment”.

7.b/3 **Discussion:** {*AI05-0041-1*} The second rule will only use the indefinite or dereference bullets in the definition of “known to be constrained”. The rule is worded in terms of “known to be constrained” in order to capture the special rules that apply in generic bodies (rather than repeating them and getting them subtly wrong).

8 A named discriminant_association with more than one selector_name is allowed only if the named discriminants are all of the same type. A discriminant_constraint shall provide exactly one value for each discriminant of the subtype being constrained.

9/3 *This paragraph was deleted.*{*AI05-0102-1*}

9.a/3 **Ramification:** In addition, 8.6 requires that the expression associated with an access discriminant is convertible (see 4.6) to the anonymous access type. This implies both convertibility of designated types, and static accessibility. This implies that if an object of type T with an access discriminant is created by an allocator for an access type A, then it requires that the type of the expression associated with the access discriminant have an accessibility level that is not statically deeper than that of A. This is to avoid dangling references.

Dynamic Semantics

A discriminant_constraint is *compatible* with an unconstrained discriminated subtype if each discriminant value belongs to the subtype of the corresponding discriminant. 10

Ramification: The "dependent compatibility check" has been eliminated in Ada 95. Any checking on subcomponents is performed when (and if) an object is created. 10.a

Discussion: There is no need to define compatibility with a constrained discriminated subtype, because one is not allowed to constrain it again. 10.b

A composite value *satisfies* a discriminant constraint if and only if each discriminant of the composite value has the value imposed by the discriminant constraint. 11

{*AI12-0439-1*} For the elaboration of a discriminant_constraint, the expressions in the discriminant_associations are evaluated in an arbitrary order and converted to the type of the associated discriminant (which can raise Constraint_Error — see 4.6); the expression of a named association is evaluated (and converted) once for each associated discriminant. The result of each evaluation and conversion is the value imposed by the constraint for the associated discriminant. 12/5

Reason: We convert to the type, not the subtype, so that the definition of compatibility of discriminant constraints is not vacuous. 12.a

NOTE The rules of the language ensure that a discriminant of an object always has a value, either from explicit or implicit initialization. 13

Discussion: Although it is illegal to constrain a class-wide tagged subtype, it is possible to have a partially constrained class-wide subtype: If the subtype S is defined by T(A => B), then S'Class is partially constrained in the sense that objects of subtype S'Class have to have discriminants corresponding to A equal to B, but there can be other discriminants defined in extensions that are not constrained to any particular value. 13.a

Examples

{AI05-0299-1} {AI12-0449-1} Examples (using types declared above in 3.7): 14/5

```
Large   : Buffer(200);   -- constrained, always 200 characters                15
                         --  (explicit discriminant value)
Message : Buffer;        -- unconstrained, initially 100 characters
                         --  (default discriminant value)
Basis   : Square(5);     -- constrained, always 5 by 5
Illegal : Square;        -- illegal, a Square has to be constrained
```

Inconsistencies With Ada 83

Dependent compatibility checks are no longer performed on subtype declaration. Instead they are deferred until object creation (see 3.3.1). This is upward compatible for a program that does not raise Constraint_Error. 15.a

Wording Changes from Ada 83

Everything in RM83-3.7.2(7-12), which specifies the initial values for discriminants, is now redundant with 3.3.1, 6.4.1, 8.5.1, and 12.4. Therefore, we don't repeat it here. Since the material is largely intuitive, but nevertheless complicated to state formally, it doesn't seem worth putting it in a "NOTE". 15.b

Incompatibilities With Ada 95

{8652/0008} {AI95-00168-01} {AI95-00363-01} The Corrigendum added a restriction on discriminant_constraints for general access subtypes. Such constraints are prohibited if the designated type can be treated as constrained somewhere in the program. Ada 2005 goes further and prohibits such discriminant_constraints if the designated type has (or might have, in the case of a formal type) defaults for its discriminants. The use of general access subtypes is rare, and this eliminates a boatload of problems that required many restrictions on the use of aliased objects and components (now lifted). Similarly, Ada 2005 prohibits discriminant_constraints on any access type whose designated type has a partial view that is constrained. Such a type will not be constrained in the heap to avoid privacy problems. Again, the use of such subtypes is rare (they can only happen within the package and its child units). 15.c/2

Wording Changes from Ada 2005

15.d/3 {*AI05-0041-1*} **Correction:** Revised the rules on access subtypes having discriminant constraints to depend on the "known to be constrained" rules. This centralizes the rules so that future fixes need to be made in only one place, as well as fixing bugs in obscure cases.

15.e/3 {*AI05-0102-1*} **Correction:** Moved implicit conversion Legality Rule to 8.6.

3.7.2 Operations of Discriminated Types

1 [If a discriminated type has default_expressions for its discriminants, then unconstrained variables of the type are permitted, and the discriminants of such a variable can be changed by assignment to the variable. For a formal parameter of such a type, an attribute is provided to determine whether the corresponding actual parameter is constrained or unconstrained.]

Static Semantics

2 For a prefix A that is of a discriminated type [(after any implicit dereference)], the following attribute is defined:

3/5 A'Constrained

{*AI05-0214-1*} {*AI12-0183-1*} Yields the value True if A denotes a constant, a value, a tagged object, or a constrained variable, and False otherwise. The value of this attribute is of the predefined type Boolean.

3.a/3 **Implementation Note:** {*AI05-0214-1*} This attribute is primarily used on parameters, to determine whether the discriminants can be changed as part of an assignment. The Constrained attribute is statically True for **in** parameters. For **in out** and **out** parameters of a discriminated type, the value of this attribute needs to be passed as an implicit parameter, in general. However, if the type is tagged or does not have defaults for its discriminants, the attribute is statically True, so no implicit parameter is needed. Parameters of a limited untagged type with defaulted discriminants need this implicit parameter, unless there are no nonlimited views, because they might be passed to a subprogram whose body has visibility on a nonlimited view of the type, and hence might be able to assign to the object and change its discriminants.

3.b/4 **Reason:** {*AI05-0214-1*} {*AI12-0005-1*} All tagged objects are known to be constrained (as nonlimited tagged types cannot have discriminant defaults, and limited tagged objects are immutably limited), and are always considered constrained by this attribute to avoid distributed overhead for parameters of limited class-wide types, as limited tagged objects may technically be unconstrained if they use defaulted discriminants. Such objects still cannot have their discriminants changed, as assignment is not supported for them, so there is no use for this attribute that would justify the overhead of passing it with all class-wide parameters.

3.c/3 **Discussion:** {*AI05-0005-1*} {*AI05-0214-1*} If the type of A is a type derived from an untagged partial view of a tagged type such that it is not a tagged type, then A is not considered a tagged object, and A'Constrained can return either True or False depending on the nature of the object.

Erroneous Execution

4 The execution of a construct is erroneous if the construct has a constituent that is a name denoting a subcomponent that depends on discriminants, and the value of any of these discriminants is changed by this execution between evaluating the name and the last use (within this execution) of the subcomponent denoted by the name.

4.a **Ramification:** This rule applies to assignment_statements, calls (except when the discriminant-dependent subcomponent is an **in** parameter passed by copy), indexed_components, and slices. Ada 83 only covered the first two cases. AI83-00585 pointed out the situation with the last two cases. The cases of object_renaming_declarations and generic formal **in out** objects are handled differently, by disallowing the situation at compile time.

Extensions to Ada 83

4.b/1 For consistency with other attributes, we are allowing the prefix of Constrained to be a value as well as an object of a discriminated type, and also an implicit dereference. These extensions are not important capabilities, but there seems no reason to make this attribute different from other similar attributes. We are curious what most Ada 83 compilers do with F(1).X'Constrained.

We now handle in a general way the cases of erroneousness identified by AI83-00585, where the prefix of an indexed_component or slice is discriminant-dependent, and the evaluation of the index or discrete range changes the value of a discriminant. 4.c

Wording Changes from Ada 83

We have moved all discussion of erroneous use of names that denote discriminant-dependent subcomponents to this subclause. In Ada 83, it used to appear separately under assignment_statements and subprogram calls. 4.d

Wording Changes from Ada 2005

{*AI05-0214-1*} A'Constrained is now defined to return True for any A that is a tagged object. This doesn't change the result for any A allowed by previous versions of Ada; the change is necessary to avoid unnecessary overhead for limited tagged parameters. 4.e/3

3.8 Record Types

A record object is a composite object consisting of named components. The value of a record object is a composite value consisting of the values of the components. 1

Syntax

record_type_definition ::= [[**abstract**] **tagged**] [**limited**] record_definition 2

{*AI12-0213-1*} record_definition ::= 3/5
```
   record
     component_list
   end record [record_identifier]
 | null record
```

component_list ::= 4
```
    component_item {component_item}
  | {component_item} variant_part
  | null;
```

{*8652/0009*} {*AI95-00137-01*} component_item ::= component_declaration | aspect_clause 5/1

{*AI05-0183-1*} component_declaration ::= 6/3
```
  defining_identifier_list : component_definition [:= default_expression]
      [aspect_specification];
```

{*AI12-0213-1*} If a *record*_identifier appears at the end of the record_definition, it shall repeat the defining_identifier of the enclosing full_type_declaration. 6.1/5

Name Resolution Rules

The expected type for the default_expression, if any, in a component_declaration is the type of the component. 7

Legality Rules

This paragraph was deleted.{*AI95-00287-01*} 8/2

{*AI95-00366-01*} Each component_declaration declares a component of the record type. Besides components declared by component_declarations, the components of a record type include any components declared by discriminant_specifications of the record type declaration. [The identifiers of all components of a record type shall be distinct.] 9/2

Proof: {*AI05-0299-1*} {*AI12-0449-1*} The identifiers of all components of a record type have to be distinct because they are all declared immediately within the same declarative region. See Clause 8. 9.a/5

10 Within a type_declaration, a name that denotes a component, protected subprogram, or entry of the type is allowed only in the following cases:

11/3 • {*AI05-0004-1*} {*AI05-0295-1*} A name that denotes any component, protected subprogram, or entry is allowed within an aspect_specification, an operational item, or a representation item that occurs within the declaration of the composite type.

12/3 • {*AI05-0264-1*} A name that denotes a noninherited discriminant is allowed within the declaration of the type, but not within the discriminant_part. If the discriminant is used to define the constraint of a component, the bounds of an entry family, or the constraint of the parent subtype in a derived_type_definition, then its name shall appear alone as a direct_name (not as part of a larger expression or expanded name). A discriminant shall not be used to define the constraint of a scalar component.

12.a **Reason:** The penultimate restriction simplifies implementation, and allows the outer discriminant and the inner discriminant or bound to possibly share storage.

12.b **Ramification:** Other rules prevent such a discriminant from being an inherited one.

12.c **Reason:** The last restriction is inherited from Ada 83. The restriction is not really necessary from a language design point of view, but we did not remove it, in order to avoid unnecessary changes to existing compilers.

12.d **Discussion:** Note that a discriminant can be used to define the constraint for a component that is of an access-to-composite type.

12.e/2 **Reason:** {*AI95-00373-01*} The above rules, and a similar one in 6.1 for formal parameters, are intended to allow initializations of components or parameters to occur in a (nearly) arbitrary order — whatever order is most efficient (subject to the restrictions of 3.3.1), since one default_expression cannot depend on the value of another one. They also prevent circularities.

12.f/3 **Ramification:** {*AI05-0295-1*} Inherited discriminants are not allowed to be denoted, except within aspect_specifications and representation items. However, the *discriminant*_selector_name of the parent subtype_indication is allowed to denote a discriminant of the parent.

13 If the name of the current instance of a type (see 8.6) is used to define the constraint of a component, then it shall appear as a direct_name that is the prefix of an attribute_reference whose result is of an access type, and the attribute_reference shall appear alone.

13.a **Reason:** This rule allows T'Access or T'Unchecked_Access, but disallows, for example, a range constraint (1..T'Size). Allowing things like (1..T'Size) would mean that a per-object constraint could affect the size of the object, which would be bad.

Static Semantics

13.1/3 {*AI95-00318-02*} {*AI05-0004-1*} If a record_type_definition includes the reserved word **limited**, the type is called an *explicitly limited record* type.

14 The component_definition of a component_declaration defines the (nominal) subtype of the component. If the reserved word **aliased** appears in the component_definition, then the component is aliased (see 3.10).

15 If the component_list of a record type is defined by the reserved word **null** and there are no discriminants, then the record type has no components and all records of the type are *null records*. A record_definition of **null record** is equivalent to **record null; end record**.

15.a/5 **Ramification:** {*AI12-0426-1*} This shorthand is available both for declaring a record type and a record extension — see 3.9.1.

Dynamic Semantics

16 The elaboration of a record_type_definition creates the record type and its first subtype, and consists of the elaboration of the record_definition. The elaboration of a record_definition consists of the elaboration of its component_list, if any.

The elaboration of a component_list consists of the elaboration of the component_items and variant_part, if any, in the order in which they appear. The elaboration of a component_declaration consists of the elaboration of the component_definition. 17

Discussion: If the defining_identifier_list has more than one defining_identifier, we presume here that the transformation explained in 3.3.1 has already taken place. Alternatively, we could say that the component_definition is elaborated once for each defining_identifier in the list. 17.a

{*8652/0002*} {*AI95-00171-01*} {*AI95-00230-01*} Within the definition of a composite type, if a component_definition or discrete_subtype_definition (see 9.5.2) includes a name that denotes a discriminant of the type, or that is an attribute_reference whose prefix denotes the current instance of the type, the expression containing the name is called a *per-object expression*, and the constraint or range being defined is called a *per-object constraint*. For the elaboration of a component_definition of a component_declaration or the discrete_subtype_definition of an entry_declaration for an entry family (see 9.5.2), if the component subtype is defined by an access_definition or if the constraint or range of the subtype_indication or discrete_subtype_definition is not a per-object constraint, then the access_definition, subtype_indication, or discrete_subtype_definition is elaborated. On the other hand, if the constraint or range is a per-object constraint, then the elaboration consists of the evaluation of any included expression that is not part of a per-object expression. Each such expression is evaluated once unless it is part of a named association in a discriminant constraint, in which case it is evaluated once for each associated discriminant. 18/2

{*8652/0002*} {*AI95-00171-01*} When a per-object constraint is elaborated [(as part of creating an object)], each per-object expression of the constraint is evaluated. For other expressions, the values determined during the elaboration of the component_definition or entry_declaration are used. Any checks associated with the enclosing subtype_indication or discrete_subtype_definition are performed[, including the subtype compatibility check (see 3.2.2),] and the associated subtype is created. 18.1/1

Discussion: The evaluation of other expressions that appear in component_definitions and discrete_subtype_definitions is performed when the type definition is elaborated. The evaluation of expressions that appear as default_expressions is postponed until an object is created. Expressions in representation items that appear within a composite type definition are evaluated according to the rules of the particular representation item. 18.a

NOTE 1 A component_declaration with several identifiers is equivalent to a sequence of single component_declarations, as explained in 3.3.1. 19

NOTE 2 The default_expression of a record component is only evaluated upon the creation of a default-initialized object of the record type (presuming the object has the component, if it is in a variant_part — see 3.3.1). 20

NOTE 3 The subtype defined by a component_definition (see 3.6) has to be a definite subtype. 21

NOTE 4 If a record type does not have a variant_part, then the same components are present in all values of the type. 22

NOTE 5 A record type is limited if it has the reserved word **limited** in its definition, or if any of its components are limited (see 7.5). 23

NOTE 6 The predefined operations of a record type include membership tests, qualification, and explicit conversion. If the record type is nonlimited, they also include assignment and the predefined equality operators. 24

NOTE 7 {*AI95-00287-01*} A component of a record can be named with a selected_component. A value of a record can be specified with a record_aggregate. 25/2

Examples

Examples of record type declarations: 26

{*AI12-0430-1*}
```
type Date is
   record
      Day   : Integer range 1 .. 31;
      Month : Month_Name;                  -- see 3.5.1
      Year  : Integer range 0 .. 4000;
   end record;
```
27/5

28/5
```
{AI12-0213-1} type Complex is
   record
      Re : Real := 0.0;
      Im : Real := 0.0;
   end record Complex;
```

29 *Examples of record variables:*

30
```
Tomorrow, Yesterday : Date;
A, B, C : Complex;
```

31
```
-- both components of A, B, and C are implicitly initialized to zero
```

Extensions to Ada 83

31.a The syntax rule for component_declaration is modified to use component_definition (instead of component_subtype_definition). The effect of this change is to allow the reserved word **aliased** before the component_subtype_definition.

31.b A short-hand is provided for defining a null record type (and a null record extension), as these will be more common for abstract root types (and derived types without additional components).

31.c The syntax rule for record_type_definition is modified to allow the reserved words **tagged** and **limited**. Tagging is new. Limitedness is now orthogonal to privateness. In Ada 83 the syntax implied that limited private was sort of more private than private. However, limitedness really has nothing to do with privateness; limitedness simply indicates the lack of assignment capabilities, and makes perfect sense for nonprivate types such as record types.

Wording Changes from Ada 83

31.d/1 {*8652/0009*} {*AI95-00137-01*} The syntax rules now allow aspect_clauses to appear in a record_definition. This is not a language extension, because Legality Rules prevent all language-defined representation clauses from appearing there. However, an implementation-defined attribute_definition_clause could appear there. The reason for this change is to allow the rules for aspect_clauses and representation pragmas to be as similar as possible.

Extensions to Ada 95

31.e/2 {*AI95-00287-01*} Record components can have an anonymous access type.

31.f/2 {*AI95-00287-01*} Limited components can be initialized, so long as the expression is one that allows building the object in place (such as an aggregate or function_call).

Wording Changes from Ada 95

31.g/2 {*8652/0002*} {*AI95-00171-01*} **Corrigendum:** Improved the description of the elaboration of per-object constraints.

31.h/2 {*8652/0009*} {*AI95-00137-01*} **Corrigendum:** Changed representation clauses to aspect clauses to reflect that they are used for more than just representation.

31.i/2 {*AI95-00318-02*} Defined *explicitly limited record* type to use in other rules.

Extensions to Ada 2005

31.j/3 {*AI05-0183-1*} An optional aspect_specification can be used in a component_declaration. This is described in 13.1.1.

Extensions to Ada 2012

31.k/5 {*AI12-0213-1*} The *record*_identifier following **end record** is new.

3.8.1 Variant Parts and Discrete Choices

1 A record type with a variant_part specifies alternative lists of components. Each variant defines the components for the value or values of the discriminant covered by its discrete_choice_list.

1.a **Discussion:** Discrete_choice_lists and discrete_choices are said to *cover* values as defined below; which discrete_choice_list covers a value determines which of various alternatives is chosen. These are used in variant_parts, array_aggregates, and case_statements.

Language Design Principles

1.b The definition of “cover” in this subclause and the rules about discrete choices are designed so that they are also appropriate for array aggregates and case statements.

The rules of this subclause intentionally parallel those for case statements. 1.c

Syntax

variant_part ::= 2
  case *discriminant_*direct_name **is**
    variant
    {variant}
  end case;

variant ::= 3
  when discrete_choice_list =>
    component_list

{*AI12-0212-1*} discrete_choice_list ::= discrete_choice {'|' discrete_choice} 4/5

{*AI05-0153-3*} {*AI05-0158-1*} discrete_choice ::= 5/3
  choice_expression | *discrete_*subtype_indication | range | **others**

Name Resolution Rules

The *discriminant_*direct_name shall resolve to denote a discriminant (called the *discriminant of the variant_part*) specified in the known_discriminant_part of the full_type_declaration that contains the variant_part. The expected type for each discrete_choice in a variant is the type of the discriminant of the variant_part. 6

Ramification: A full_type_declaration with a variant_part has to have a (new) known_discriminant_part; the discriminant of the variant_part cannot be an inherited discriminant. 6.a

Legality Rules

The discriminant of the variant_part shall be of a discrete type. 7

Ramification: It shall not be of an access type, named or anonymous. 7.a

{*AI05-0153-3*} The choice_expressions, subtype_indications, and ranges given as discrete_choices in a variant_part shall be static. The discrete_choice **others** shall appear alone in a discrete_choice_list, and such a discrete_choice_list, if it appears, shall be the last one in the enclosing construct. 8/3

A discrete_choice is defined to *cover a value* in the following cases: 9

- {*AI05-0262-1*} A discrete_choice that is a choice_expression covers a value if the value equals the value of the choice_expression converted to the expected type. 10/3
- {*AI05-0153-3*} {*AI05-0262-1*} {*AI12-0071-1*} A discrete_choice that is a subtype_indication covers all values (possibly none) that belong to the subtype and that satisfy the static predicates of the subtype (see 3.2.4). 10.1/4

  Ramification: {*AI05-0262-1*} A dynamic predicate is never allowed in this case (for variants, case_statements, and case_expressions, a subtype with a dynamic predicate isn't static and thus isn't allowed in a discrete_choice, and for a choice in an array_aggregate, a dynamic predicate is explicitly disallowed — see 3.2.4). 10.a/3
- {*AI05-0153-3*} A discrete_choice that is a range covers all values (possibly none) that belong to the range. 11/3
- The discrete_choice **others** covers all values of its expected type that are not covered by previous discrete_choice_lists of the same construct. 12

  Ramification: For case_statements, this includes values outside the range of the static subtype (if any) to be covered by the choices. It even includes values outside the base range of the case expression's type, since values of numeric types (and undefined values of any scalar type?) can be outside their base range. 12.a

A discrete_choice_list covers a value if one of its discrete_choices covers the value. 13

14 The possible values of the discriminant of a variant_part shall be covered as follows:

15/4 • {*AI05-0153-3*} {*AI05-0188-1*} {*AI05-0262-1*} {*AI12-0071-1*} If the discriminant is of a static constrained scalar subtype then, except within an instance of a generic unit, each non-**others** discrete_choice shall cover only values in that subtype that satisfy its predicates, and each value of that subtype that satisfies its predicates shall be covered by some discrete_choice [(either explicitly or by **others**)];

15.a/3 **Reason:** {*AI05-0188-1*} The exemption for a discriminated type declared in an instance allows the following example:

15.b/3
```
generic
   type T is new Integer;
package G is
   type Rec (Discrim : T) is record
      case Discrim is
         when -10 .. -1 =>
            Foo : Float;
         when others =>
            null;
      end case;
   end record;
end G;
```

15.c/3
```
package I is new G (Natural); -- Legal
```

16/3 • {*AI05-0264-1*} If the type of the discriminant is a descendant of a generic formal scalar type, then the variant_part shall have an **others** discrete_choice;

16.a **Reason:** The base range is not known statically in this case.

17 • Otherwise, each value of the base range of the type of the discriminant shall be covered [(either explicitly or by **others**)].

18 Two distinct discrete_choices of a variant_part shall not cover the same value.

Static Semantics

19 If the component_list of a variant is specified by **null**, the variant has no components.

20 The discriminant of a variant_part is said to *govern* the variant_part and its variants. In addition, the discriminant of a derived type governs a variant_part and its variants if it corresponds (see 3.7) to the discriminant of the variant_part.

Dynamic Semantics

21 A record value contains the values of the components of a particular variant only if the value of the discriminant governing the variant is covered by the discrete_choice_list of the variant. This rule applies in turn to any further variant that is, itself, included in the component_list of the given variant.

21.1/3 {*AI05-0290-1*} When an object of a discriminated type *T* is initialized by default, Constraint_Error is raised if no discrete_choice_list of any variant of a variant_part of *T* covers the value of the discriminant that governs the variant_part. When a variant_part appears in the component_list of another variant *V*, this test is only applied if the value of the discriminant governing *V* is covered by the discrete_choice_list of *V*.

21.a/3 **Implementation Note:** This is not a "check"; it cannot be suppressed. However, in most cases it is not necessary to generate any code to raise this exception. A test is needed (and can fail) in the case where the discriminant subtype has a Static_Predicate specified, it also has predicate checking disabled, and the discriminant governs a variant_part which lacks a **when others** choice.

21.b/3 The test also could fail for a static discriminant subtype with range checking suppressed and the discriminant governs a variant_part which lacks a **when others** choice. But execution is erroneous if a range check that would have failed is suppressed (see 11.5), so an implementation does not have to generate code to check this case. (An unchecked failed predicate does not cause erroneous execution, so the test is required in that case.)

21.c/3 Like the checks associated with a per-object constraint, this test is not made during the elaboration of a subtype_indication.

The elaboration of a variant_part consists of the elaboration of the component_list of each variant in the order in which they appear. 22

Examples

Example of record type with a variant part: 23

```
type Device is (Printer, Disk, Drum);                                    24
type State  is (Open, Closed);

type Peripheral(Unit : Device := Disk) is                                25
   record
      Status : State;
      case Unit is
         when Printer =>
            Line_Count : Integer range 1 .. Page_Size;
         when others =>
            Cylinder   : Cylinder_Index;
            Track      : Track_Number;
      end case;
   end record;
```

Examples of record subtypes: 26

```
subtype Drum_Unit is Peripheral(Drum);                                   27
subtype Disk_Unit is Peripheral(Disk);
```

Examples of constrained record variables: 28

```
Writer   : Peripheral(Unit  => Printer);                                 29
Archive  : Disk_Unit;
```

Extensions to Ada 83

In Ada 83, the discriminant of a variant_part is not allowed to be of a generic formal type. This restriction is removed in Ada 95; an **others** discrete_choice is required in this case. 29.a

Wording Changes from Ada 83

The syntactic category choice is removed. The syntax rules for variant, array_aggregate, and case_statement now use discrete_choice_list or discrete_choice instead. The syntax rule for record_aggregate now defines its own syntax for named associations. 29.b

{*AI05-0299-1*} We have added the term Discrete Choice to the title since this is where they are talked about. This is analogous to the name of the subclause "Index Constraints and Discrete Ranges" in the subclause on Array Types. 29.c/3

The rule requiring that the discriminant denote a discriminant of the type being defined seems to have been left implicit in RM83. 29.d

Incompatibilities With Ada 2005

{*AI05-0158-1*} Membership tests are no longer allowed as a discrete_choice, in order that those tests can be expanded to allow multiple tests in a single expression without ambiguity. Since a membership test has a boolean type, they are very unlikely to be used as a discrete_choice. 29.e/3

Extensions to Ada 2005

{*AI05-0153-3*} Subtypes with static predicates can be used in discrete_choices, and the coverage rules are modified to respect the predicates. 29.f/3

{*AI05-0188-1*} Variants in generic specifications are no longer rejected if the subtype of the actual type does not include all of the case choices. This probably isn't useful, but it is consistent with the treatment of case_expressions. 29.g/3

Wording Changes from Ada 2005

{*AI05-0290-1*} Added a test that some variant covers the value of a discriminant that governs a variant_part. This is similar to the test that some case limb covers the value of the *Selecting_*expression of a case_statement. This test cannot change the behavior of any nonerroneous Ada 2005 program, so it is not an inconsistency. 29.h/3

Wording Changes from Ada 2012

29.i/4 {*AI12-0071-1*} **Corrigendum:** Updated wording of the coverage rules to use the new term "satisfies the predicates" (see 3.2.4).

3.9 Tagged Types and Type Extensions

1 [Tagged types and type extensions support object-oriented programming, based on inheritance with extension and run-time polymorphism via *dispatching operations*.]

Language Design Principles

1.a/2 {*AI95-00251-01*} The intended implementation model is for the static portion of a tag to be represented as a pointer to a statically allocated and link-time initialized type descriptor. The type descriptor contains the address of the code for each primitive operation of the type. It probably also contains other information, such as might make membership tests convenient and efficient. Tags for nested type extensions must also have a dynamic part that identifies the particular elaboration of the type.

1.b The primitive operations of a tagged type are known at its first freezing point; the type descriptor is laid out at that point. It contains linker symbols for each primitive operation; the linker fills in the actual addresses.

1.b.1/2 {*AI95-00251-01*} Primitive operations of type extensions that are declared at a level deeper than the level of the ultimate ancestor from which they are derived can be represented by wrappers that use the dynamic part of the tag to call the actual primitive operation. The dynamic part would generally be some way to represent the static link or display necessary for making a nested call. One implementation strategy would be to store that information in the extension part of such nested type extensions, and use the dynamic part of the tag to point at it. (That way, the "dynamic" part of the tag could be static, at the cost of indirect access.)

1.b.2/2 {*AI95-00251-01*} If the tagged type is descended from any interface types, it also will need to include "subtags" (one for each interface) that describe the mapping of the primitive operations of the interface to the primitives of the type. These subtags could directly reference the primitive operations (for faster performance), or simply provide the tag "slot" numbers for the primitive operations (for easier derivation). In either case, the subtags would be used for calls that dispatch through a class-wide type of the interface.

1.c Other implementation models are possible.

1.d The rules ensure that "dangling dispatching" is impossible; that is, when a dispatching call is made, there is always a body to execute. This is different from some other object-oriented languages, such as Smalltalk, where it is possible to get a run-time error from a missing method.

1.e/2 {*AI95-00251-01*} Dispatching calls should be efficient, and should have a bounded worst-case execution time. This is important in a language intended for real-time applications. In the intended implementation model, a dispatching call involves calling indirect through the appropriate slot in the dispatch table. No complicated "method lookup" is involved although a call which is dispatching on an interface may require a lookup of the appropriate interface subtag.

1.f The programmer should have the choice at each call site of a dispatching operation whether to do a dispatching call or a statically determined call (i.e. whether the body executed should be determined at run time or at compile time).

1.g The same body should be executed for a call where the tag is statically determined to be T'Tag as for a dispatching call where the tag is found at run time to be T'Tag. This allows one to test a given tagged type with statically determined calls, with some confidence that run-time dispatching will produce the same behavior.

1.h All views of a type should share the same type descriptor and the same tag.

1.i The visibility rules determine what is legal at compile time; they have nothing to do with what bodies can be executed at run time. Thus, it is possible to dispatch to a subprogram whose declaration is not visible at the call site. In fact, this is one of the primary facts that gives object-oriented programming its power. The subprogram that ends up being dispatched to by a given call might even be designed long after the call site has been coded and compiled.

1.j Given that Ada has overloading, determining whether a given subprogram overrides another is based both on the names and the type profiles of the operations.

1.k/2 {*AI95-00401-01*} When a type extension is declared, if there is any place within its immediate scope where a certain subprogram of the parent or progenitor is visible, then a matching subprogram should override. If there is no such place, then a matching subprogram should be totally unrelated, and occupy a different slot in the type descriptor. This is important to preserve the privacy of private parts; when an operation declared in a private part is inherited, the inherited version can be overridden only in that private part, in the package body, and in any children of the package.

1.l If an implementation shares code for instances of generic bodies, it should be allowed to share type descriptors of tagged types declared in the generic body, so long as they are not extensions of types declared in the specification of the generic unit.

Static Semantics

{*AI95-00345-01*} A record type or private type that has the reserved word **tagged** in its declaration is called a *tagged* type. In addition, an interface type is a tagged type, as is a task or protected type derived from an interface (see 3.9.4). [When deriving from a tagged type, as for any derived type, additional primitive subprograms may be defined, and inherited primitive subprograms may be overridden.] The derived type is called an *extension* of its ancestor types, or simply a *type extension*. 2/2

{*AI95-00345-01*} Every type extension is also a tagged type, and is a *record extension* or a *private extension* of some other tagged type, or a noninterface synchronized tagged type (see 3.9.4). A record extension is defined by a derived_type_definition with a record_extension_part (see 3.9.1)[, which may include the definition of additional components]. A private extension, which is a partial view of a record extension or of a synchronized tagged type, can be declared in the visible part of a package (see 7.3) or in a generic formal part (see 12.5.1). 2.1/2

Term entry: tagged type — type whose objects each have a run-time type tag, which indicates the specific type for which the object was originally created 2.a/5
Note: Tagged types can be extended with additional components.

Ramification: {*AI95-00218-03*} If a tagged type is declared other than in a package_specification, it is impossible to add new primitive subprograms for that type, although it can inherit primitive subprograms, and those can be overridden. If the user incorrectly thinks a certain subprogram is primitive when it is not, and tries to call it with a dispatching call, an error message will be given at the call site. Similarly, by using an overriding_indicator (see 6.1), the user can declare that a subprogram is intended to be overriding, and get an error message when they made a mistake. The use of overriding_indicators is highly recommended in new code that does not need to be compatible with Ada 95. 2.b/2

An object of a tagged type has an associated (run-time) *tag* that identifies the specific tagged type used to create the object originally. [The tag of an operand of a class-wide tagged type *T*'Class controls which subprogram body is to be executed when a primitive subprogram of type *T* is applied to the operand (see 3.9.2); using a tag to control which body to execute is called *dispatching*.] 3

{*AI95-00344-01*} The tag of a specific tagged type identifies the full_type_declaration of the type, and for a type extension, is sufficient to uniquely identify the type among all descendants of the same ancestor. If a declaration for a tagged type occurs within a generic_package_declaration, then the corresponding type declarations in distinct instances of the generic package are associated with distinct tags. For a tagged type that is local to a generic package body and with all of its ancestors (if any) also local to the generic body, the language does not specify whether repeated instantiations of the generic body result in distinct tags. 4/2

This paragraph was deleted.{*AI95-00344-01*} 4.a/2

Implementation Note: {*AI95-00344-01*} In most cases, a tag need only identify a particular tagged type declaration, and can therefore be a simple link-time-known address. However, for tag checks (see 3.9.2) it is essential that each descendant (that currently exists) of a given type have a unique tag. Hence, for types declared in shared generic bodies where an ancestor comes from outside the generic, or for types declared at a deeper level than an ancestor, the tag needs to be augmented with some kind of dynamic descriptor (which may be a static link, global display, instance descriptor pointer, or combination). This implies that type Tag may need to be two words, the second of which is normally null, but in these identified special cases needs to include a static link or equivalent. Within an object of one of these types with a two-word tag, the two parts of the tag would typically be separated, one part as the first word of the object, the second placed in the first extension part that corresponds to a type declared more nested than its parent or declared in a shared generic body when the parent is declared outside. Alternatively, by using an extra level of indirection, the type Tag could remain a single-word. 4.a.1/2

{*AI95-00344-01*} For types that are not type extensions (even for ones declared in nested scopes), we do not require that repeated elaborations of the same full_type_declaration correspond to distinct tags. This was done so that Ada 2005 implementations of tagged types could maintain representation compatibility with Ada 95 implementations. Only type extensions that were not allowed in Ada 95 require additional information with the tag. 4.b/2

To be honest: {*AI95-00344-01*} The wording "is sufficient to uniquely identify the type among all descendants of the same ancestor" only applies to types that currently exist. It is not necessary to distinguish between descendants that currently exist, and descendants of the same type that no longer exist. For instance, the address of the stack frame of 4.c/2

the subprogram that created the tag is sufficient to meet the requirements of this rule, even though it is possible, after the subprogram returns, that a later call of the subprogram could have the same stack frame and thus have an identical tag.

5 The following language-defined library package exists:

6/5
```
{AI95-00362-01} {AI12-0241-1} {AI12-0302-1} {AI12-0399-1} package Ada.Tags
    with Preelaborate, Nonblocking, Global => in out synchronized is
    type Tag is private
       with Preelaborable_Initialization;
```

6.1/2
```
{AI95-00260-02}     No_Tag : constant Tag;
```

7/2
```
{AI95-00400-01}     function Expanded_Name(T : Tag) return String;
    function Wide_Expanded_Name(T : Tag) return Wide_String;
    function Wide_Wide_Expanded_Name(T : Tag) return Wide_Wide_String;
    function External_Tag(T : Tag) return String;
    function Internal_Tag(External : String) return Tag;
```

7.1/2
```
{AI95-00344-01}     function Descendant_Tag(External : String;
                                Ancestor : Tag) return Tag;
    function Is_Descendant_At_Same_Level(Descendant, Ancestor : Tag)
        return Boolean;
```

7.2/2
```
{AI95-00260-02}     function Parent_Tag (T : Tag) return Tag;
```

7.3/2
```
{AI95-00405-01}     type Tag_Array is array (Positive range <>) of Tag;
```

7.4/2
```
{AI95-00405-01}     function Interface_Ancestor_Tags (T : Tag) return Tag_Array;
```

7.5/3
```
{AI05-0173-1}     function Is_Abstract (T : Tag) return Boolean;
```

8
```
    Tag_Error : exception;
```

9
```
private
   ... -- not specified by the language
end Ada.Tags;
```

9.a **Reason:** Tag is a nonlimited, definite subtype, because it needs the equality operators, so that tag checking makes sense. Also, equality, assignment, and object declaration are all useful capabilities for this subtype.

9.b For an object X and a type T, “X'Tag = T'Tag” is not needed, because a membership test can be used. However, comparing the tags of two objects cannot be done via membership. This is one reason to allow equality for type Tag.

9.1/2 {*AI95-00260-02*} No_Tag is the default initial value of type Tag.

9.c/2 **Reason:** {*AI95-00260-02*} This is similar to the requirement that all access values be initialized to **null**.

10/2 {*AI95-00400-01*} The function Wide_Wide_Expanded_Name returns the full expanded name of the first subtype of the specific type identified by the tag, in upper case, starting with a root library unit. The result is implementation defined if the type is declared within an unnamed block_statement.

10.a **To be honest:** This name, as well as each prefix of it, does not denote a renaming_declaration.

10.b/2 **Implementation defined:** The result of Tags.Wide_Wide_Expanded_Name for types declared within an unnamed block_statement.

10.1/2 {*AI95-00400-01*} The function Expanded_Name (respectively, Wide_Expanded_Name) returns the same sequence of graphic characters as that defined for Wide_Wide_Expanded_Name, if all the graphic characters are defined in Character (respectively, Wide_Character); otherwise, the sequence of characters is implementation defined, but no shorter than that returned by Wide_Wide_Expanded_Name for the same value of the argument.

10.c/2 **Implementation defined:** The sequence of characters of the value returned by Tags.Expanded_Name (respectively, Tags.Wide_Expanded_Name) when some of the graphic characters of Tags.Wide_Wide_Expanded_Name are not defined in Character (respectively, Wide_Character).

11 The function External_Tag returns a string to be used in an external representation for the given tag. The call External_Tag(S'Tag) is equivalent to the attribute_reference S'External_Tag (see 13.3).

11.a **Reason:** It might seem redundant to provide both the function External_Tag and the attribute External_Tag. The function is needed because the attribute can't be applied to values of type Tag. The attribute is needed so that it can be specified via an attribute_definition_clause.

{*AI95-00417-01*} The string returned by the functions Expanded_Name, Wide_Expanded_Name, Wide_Wide_Expanded_Name, and External_Tag has lower bound 1. 11.1/2

{*AI95-00279-01*} The function Internal_Tag returns a tag that corresponds to the given external tag, or raises Tag_Error if the given string is not the external tag for any specific type of the partition. Tag_Error is also raised if the specific type identified is a library-level type whose tag has not yet been created (see 13.14). 12/2

Reason: {*AI95-00279-01*} {*AI05-0005-1*} The check for uncreated library-level types prevents a reference to the type before execution reaches the freezing point of the type. This is important so that T'Class'Input or an instance of Tags.Generic_Dispatching_Constructor do not try to create an object of a type that hasn't been frozen (which might not have yet elaborated its constraints). We don't require this behavior for non-library-level types as the tag can be created multiple times and possibly multiple copies can exist at the same time, making the check complex. 12.a/3

{*AI95-00344-01*} {*AI05-0113-1*} The function Descendant_Tag returns the (internal) tag for the type that corresponds to the given external tag and is both a descendant of the type identified by the Ancestor tag and has the same accessibility level as the identified ancestor. Tag_Error is raised if External is not the external tag for such a type. Tag_Error is also raised if the specific type identified is a library-level type whose tag has not yet been created, or if the given external tag identifies more than one type that has the appropriate Ancestor and accessibility level. 12.1/3

Reason: Descendant_Tag is used by T'Class'Input to identify the type identified by an external tag. Because there can be multiple elaborations of a given type declaration, Internal_Tag does not have enough information to choose a unique such type. Descendant_Tag does not return the tag for types declared at deeper accessibility levels than the ancestor because there could be ambiguity in the presence of recursion or multiple tasks. Descendant_Tag can be used in constructing a user-defined replacement for T'Class'Input. 12.b/2

{*AI05-0113-1*} Rules for specifying external tags will usually prevent an external tag from identifying more than one type. However, an external tag can identify multiple types if a generic body contains a derivation of a tagged type declared outside of the generic, and there are multiple instances at the same accessibility level as the type. (The Reference Manual allows default external tags to not be unique in this case.) 12.b.1/3

{*AI95-00344-01*} The function Is_Descendant_At_Same_Level returns True if the Descendant tag identifies a type that is both a descendant of the type identified by Ancestor and at the same accessibility level. If not, it returns False. 12.2/2

Reason: Is_Descendant_At_Same_Level (or something similar to it) is used by T'Class'Output to determine whether the item being written is at the same accessibility level as T. It may be used to determine prior to using T'Class'Output whether Tag_Error will be raised, and also can be used in constructing a user-defined replacement for T'Class'Output. 12.c/2

{*AI05-0115-1*} For the purposes of the dynamic semantics of functions Descendant_Tag and Is_Descendant_At_Same_Level, a tagged type T2 is a *descendant* of a type T1 if it is the same as T1, or if its parent type or one of its progenitor types is a descendant of type T1 by this rule[, even if at the point of the declaration of T2, one of the derivations in the chain is not visible]. 12.3/3

Discussion: In other contexts, “descendant” is dependent on visibility, and the particular view a derived type has of its parent type. See 7.3.1. 12.c.1/3

{*AI95-00260-02*} {*AI12-0056-1*} The function Parent_Tag returns the tag of the parent type of the type whose tag is T. If the type does not have a parent type (that is, it was not defined by a derived_type_definition), then No_Tag is returned. 12.4/4

Ramification: {*AI12-0005-1*} The parent type is always the parent of the full type; a private extension appears to define a parent type, but it does not (only the various forms of derivation do that). As this is a run-time operation, ignoring privacy is OK. 12.d/4

{*AI95-00405-01*} The function Interface_Ancestor_Tags returns an array containing the tag of each interface ancestor type of the type whose tag is T, other than T itself. The lower bound of the returned array is 1, and the order of the returned tags is unspecified. Each tag appears in the result exactly once.[If the type whose tag is T has no interface ancestors, a null array is returned.] 12.5/3

Ramification: The result of Interface_Ancestor_Tags includes the tag of the parent type, if the parent is an interface. 12.e/2

12.f/2 Indirect interface ancestors are included in the result of Interface_Ancestor_Tags. That's because where an interface appears in the derivation tree has no effect on the semantics of the type; the only interesting property is whether the type has an interface as an ancestor.

12.6/3 {*AI05-0173-1*} The function Is_Abstract returns True if the type whose tag is T is abstract, and False otherwise.

13 For every subtype S of a tagged type *T* (specific or class-wide), the following attributes are defined:

14 S'Class S'Class denotes a subtype of the class-wide type (called *T*'Class in this document) for the class rooted at *T* (or if S already denotes a class-wide subtype, then S'Class is the same as S).

15 S'Class is unconstrained. However, if S is constrained, then the values of S'Class are only those that when converted to the type *T* belong to S.

15.a **Ramification:** This attribute is defined for both specific and class-wide subtypes. The definition is such that S'Class'Class is the same as S'Class.

15.b Note that if S is constrained, S'Class is only partially constrained, since there might be additional discriminants added in descendants of *T* which are not constrained.

15.c/2 **Reason:** {*AI95-00326-01*} The Class attribute is not defined for untagged subtypes (except for incomplete types and private types whose full view is tagged — see J.11 and 7.3.1) so as to preclude implicit conversion in the absence of run-time type information. If it were defined for untagged subtypes, it would correspond to the concept of universal types provided for the predefined numeric classes.

16 S'Tag S'Tag denotes the tag of the type *T* (or if *T* is class-wide, the tag of the root type of the corresponding class). The value of this attribute is of type Tag.

16.a **Reason:** S'Class'Tag equals S'Tag, to avoid generic contract model problems when S'Class is the actual type associated with a generic formal derived type.

17 Given a prefix X that is of a class-wide tagged type [(after any implicit dereference)], the following attribute is defined:

18 X'Tag X'Tag denotes the tag of X. The value of this attribute is of type Tag.

18.a **Reason:** X'Tag is not defined if X is of a specific type. This is primarily to avoid confusion that might result about whether the Tag attribute should reflect the tag of the type of X, or the tag of X. No such confusion is possible if X is of a class-wide type.

18.1/2 {*AI95-00260-02*} {*AI95-00441-01*} The following language-defined generic function exists:

18.2/5
```
{AI05-0229-1} {AI12-0241-1} {AI12-0302-1} generic
    type T (<>) is abstract tagged limited private;
    type Parameters (<>) is limited private;
    with function Constructor (Params : not null access Parameters)
        return T is abstract;
function Ada.Tags.Generic_Dispatching_Constructor
   (The_Tag : Tag;
    Params  : not null access Parameters) return T'Class
   with Preelaborate, Convention => Intrinsic,
        Nonblocking, Global => in out synchronized;
```

18.3/2 {*AI95-00260-02*} Tags.Generic_Dispatching_Constructor provides a mechanism to create an object of an appropriate type from just a tag value. The function Constructor is expected to create the object given a reference to an object of type Parameters.

18.b/2 **Discussion:** This specification is designed to make it easy to create dispatching constructors for streams; in particular, this can be used to construct overridings for T'Class'Input.

18.c/5 {*AI12-0005-1*} Note that almost any tagged type can be used in an instance of Generic_Dispatching_Constructor. Using a tagged incomplete view or a tagged partial view before the completion of the type in such an instance would be illegal; all other tagged types can be used in an instance of Generic_Dispatching_Constructor.

Dynamic Semantics

The tag associated with an object of a tagged type is determined as follows: 19

- The tag of a stand-alone object, a component, or an aggregate of a specific tagged type *T* identifies *T*. 20

  Discussion: The tag of a formal parameter of type *T* is not necessarily the tag of *T*, if, for example, the actual was a type conversion. 20.a

- {*AI12-0437-1*} The tag of an object created by an allocator for an access type with a specific designated tagged type *T* identifies *T*. 21/5

  Discussion: The tag of an object designated by a value of such an access type might not be *T*, if, for example, the access value is the result of a type conversion. 21.a

- The tag of an object of a class-wide tagged type is that of its initialization expression. 22

  Ramification: The tag of an object (even a class-wide one) cannot be changed after it is initialized, since a "class-wide" assignment_statement raises Constraint_Error if the tags don't match, and a "specific" assignment_statement does not affect the tag. 22.a

- The tag of the result returned by a function whose result type is a specific tagged type *T* identifies *T*. 23

  Implementation Note: {*AI95-00318-02*} For a limited tagged type, the return object is "built in place" in the ultimate result object with the appropriate tag. For a nonlimited type, a new anonymous object with the appropriate tag is created as part of the function return. See 6.5, "Return Statements". 23.a/2

- {*AI95-00318-02*} The tag of the result returned by a function with a class-wide result type is that of the return object. 24/2

The tag is preserved by type conversion and by parameter passing. The tag of a value is the tag of the associated object (see 6.2). 25

{*AI95-00260-02*} {*AI95-00344-01*} {*AI95-00405-01*} {*AI05-0092-1*} {*AI05-0262-1*} Tag_Error is raised by a call of Descendant_Tag, Expanded_Name, External_Tag, Interface_Ancestor_Tags, Is_Abstract, Is_Descendant_At_Same_Level, Parent_Tag, Wide_Expanded_Name, or Wide_Wide_Expanded_Name if any tag passed is No_Tag. 25.1/3

{*AI95-00260-02*} An instance of Tags.Generic_Dispatching_Constructor raises Tag_Error if The_Tag does not represent a concrete descendant of T or if the innermost master (see 7.6.1) of this descendant is not also a master of the instance. Otherwise, it dispatches to the primitive function denoted by the formal Constructor for the type identified by The_Tag, passing Params, and returns the result. Any exception raised by the function is propagated. 25.2/2

Ramification: The tag check checks both that The_Tag is in T'Class, and that it is not abstract. These checks are similar to the ones required by streams for T'Class'Input (see 13.13.2). In addition, there is a check that the tag identifies a type declared on the current dynamic call chain, and not a more nested type or a type declared by another task. This check is not necessary for streams, because the stream attributes are declared at the same dynamic level as the type used. 25.a/2

Erroneous Execution

{*AI95-00260-02*} If an internal tag provided to an instance of Tags.Generic_Dispatching_Constructor or to any subprogram declared in package Tags identifies either a type that is not library-level and whose tag has not been created (see 13.14), or a type that does not exist in the partition at the time of the call, then execution is erroneous. 25.3/2

Ramification: One reason that a type might not exist in the partition is that the tag refers to a type whose declaration was elaborated as part of an execution of a subprogram_body which has been left (see 7.6.1). 25.b/2

We exclude tags of library-level types from the current execution of the partition, because misuse of such tags should always be detected. T'Tag freezes the type (and thus creates the tag), and Internal_Tag and Descendant_Tag cannot return the tag of a library-level type that has not been created. All ancestors of a tagged type must be frozen no later than the (full) declaration of a type that uses them, so Parent_Tag and Interface_Ancestor_Tags cannot return a tag that 25.c/2

has not been created. Finally, library-level types never cease to exist while the partition is executing. Thus, if the tag comes from a library-level type, there cannot be erroneous execution (the use of Descendant_Tag rather than Internal_Tag can help ensure that the tag is of a library-level type). This is also similar to the rules for T'Class'Input (see 13.13.2).

25.d/2 **Discussion:** {*AI95-00344-01*} Ada 95 allowed Tag_Error in this case, or expected the functions to work. This worked because most implementations used tags constructed at link-time, and each elaboration of the same type_declaration produced the same tag. However, Ada 2005 requires at least part of the tags to be dynamically constructed for a type derived from a type at a shallower level. For dynamically constructed tags, detecting the error can be expensive and unreliable. To see this, consider a program containing two tasks. Task A creates a nested tagged type, passes the tag to task B (which saves it), and then terminates. The nested tag (if dynamic) probably will need to refer in some way to the stack frame for task A. If task B later tries to use the tag created by task A, the tag's reference to the stack frame of A probably is a dangling pointer. Avoiding this would require some sort of protected tag manager, which would be a bottleneck in a program's performance. Moreover, we'd still have a race condition; if task A terminated after the tag check, but before the tag was used, we'd still have a problem. That means that all of these operations would have to be serialized. That could be a significant performance drain, whether or not nested tagged types are ever used. Therefore, we allow execution to become erroneous as we do for other dangling pointers. If the implementation can detect the error, we recommend that Tag_Error be raised.

Implementation Permissions

26/2 {*AI95-00260-02*} {*AI95-00279-01*} The implementation of Internal_Tag and Descendant_Tag may raise Tag_Error if no specific type corresponding to the string External passed as a parameter exists in the partition at the time the function is called, or if there is no such type whose innermost master is a master of the point of the function call.

26.a/2 **Reason:** {*AI95-00260-02*} {*AI95-00279-01*} {*AI95-00344-01*} Locking would be required to ensure that the mapping of strings to tags never returned tags of types which no longer exist, because types can cease to exist (because they belong to another task, as described above) during the execution of these operations. Moreover, even if these functions did use locking, that would not prevent the type from ceasing to exist at the instant that the function returned. Thus, we do not require the overhead of locking; hence the word “may” in this rule.

Implementation Advice

26.1/3 {*AI95-00260-02*} {*AI05-0113-1*} Internal_Tag should return the tag of a type, if one exists, whose innermost master is a master of the point of the function call.

26.b/3 **Implementation Advice:** Tags.Internal_Tag should return the tag of a type, if one exists, whose innermost master is a master of the point of the function call.

26.c/2 **Reason:** {*AI95-00260-02*} {*AI95-00344-01*} It's not helpful if Internal_Tag returns the tag of some type in another task when one is available in the task that made the call. We don't require this behavior (because it requires the same implementation techniques we decided not to insist on previously), but encourage it.

26.d/3 **Discussion:** {*AI05-0113-1*} There is no Advice for the result of Internal_Tag if no such type exists. In most cases, the Implementation Permission can be used to raise Tag_Error, but some other tag can be returned as well.

27/5 NOTE 1 {*AI12-0442-1*} A type declared with the reserved word **tagged** is normally declared in a package_specification, so that new primitive subprograms can be declared for it.

28 NOTE 2 Once an object has been created, its tag never changes.

29/5 NOTE 3 {*AI12-0447-1*} Class-wide types are defined to have unknown discriminants (see 3.7). This means that, by the rules in 3.7 for objects with unknown discriminants, objects of a class-wide type are illegal unless they are explicitly initialized (whether created by an object_declaration or an allocator), and that aggregates are illegal unless they are explicitly qualified with a specific type when their expected type is class-wide.

30/2 NOTE 4 {*AI95-00260-02*} {*AI95-00326-01*} The capability provided by Tags.Generic_Dispatching_Constructor is sometimes known as a *factory*.

Examples

31 *Examples of tagged record types:*

32
```
type Point is tagged
  record
    X, Y : Real := 0.0;
  end record;
```

```
type Expression is tagged null record;                                    33
  -- Components will be added by each extension
```

Extensions to Ada 83

Tagged types are a new concept. 33.a

Inconsistencies With Ada 95

{*AI95-00279-01*} **Amendment Correction:** Added wording specifying that Internal_Tag must raise Tag_Error if the tag of a library-level type has not yet been created. Ada 95 gave an Implementation Permission to do this; we require it to avoid erroneous execution when streaming in an object of a library-level type that has not yet been elaborated. This is technically inconsistent; a program that used Internal_Tag outside of streaming and used a compiler that didn't take advantage of the Implementation Permission would not have raised Tag_Error, and may have returned a useful tag. (If the tag was used in streaming, the program would have been erroneous.) Since such a program would not have been portable to a compiler that did take advantage of the Implementation Permission, this is not a significant inconsistency. 33.b/2

{*AI95-00417-01*} We now define the lower bound of the string returned from [[Wide_]Wide_]Expanded_Name and External_Name. This makes working with the returned string easier, and is consistent with many other string-returning functions in Ada. This is technically an inconsistency; if a program depended on some other lower bound for the string returned from one of these functions, it could fail when compiled with Ada 2005. Such code is not portable even between Ada 95 implementations, so it should be very rare. 33.c/2

Incompatibilities With Ada 95

{*AI95-00260-02*} {*AI95-00344-01*} {*AI95-00400-01*} {*AI95-00405-01*} {*AI05-0005-1*} Constant No_Tag, and functions Parent_Tag, Interface_Ancestor_Tags, Descendant_Tag, Is_Descendant_At_Same_Level, Wide_Expanded_Name, and Wide_Wide_Expanded_Name are added to Ada.Tags. If Ada.Tags is referenced in a use_clause, and an entity *E* with the same defining_identifier as a new entity in Ada.Tags is defined in a package that is also referenced in a use_clause, the entity *E* may no longer be use-visible, resulting in errors. This should be rare and is easily fixed if it does occur. 33.d/3

Extensions to Ada 95

{*AI95-00362-01*} Ada.Tags is now defined to be preelaborated. 33.e/2

{*AI95-00260-02*} Generic function Tags.Generic_Dispatching_Constructor is new. 33.f/2

Wording Changes from Ada 95

{*AI95-00318-02*} We talk about return objects rather than return expressions, as functions can return using an extended_return_statement. 33.g/2

{*AI95-00344-01*} Added wording to define that tags for all descendants of a tagged type must be distinct. This is needed to ensure that more nested type extensions will work properly. The wording does not require implementation changes for types that were allowed in Ada 95. 33.h/2

Inconsistencies With Ada 2005

{*AI05-0113-1*} **Correction:** Added wording specifying that Dependent_Tag must raise Tag_Error if there is more than one type which matches the requirements. If an implementation had returned a random tag of the matching types, a program may have worked properly. However, such a program would not be portable (another implementation may return a different tag) and the conditions that would cause the problem are unlikely (most likely, a tagged type extension declared in a generic body with multiple instances in the same scope). 33.i/3

Incompatibilities With Ada 2005

{*AI05-0173-1*} Function Is_Abstract is added to Ada.Tags. If Ada.Tags is referenced in a use_clause, and an entity *E* with the defining_identifier Is_Abstract is defined in a package that is also referenced in a use_clause, the entity *E* may no longer be use-visible, resulting in errors. This should be rare and is easily fixed if it does occur. 33.j/3

Wording Changes from Ada 2005

{*AI05-0115-1*} **Correction:** We explicitly define the meaning of "descendant" at runtime, so that it does not depend on visibility as does the usual meaning. 33.k/3

3.9.1 Type Extensions

1/2 {*AI95-00345-01*} [Every type extension is a tagged type, and is a *record extension* or a *private extension* of some other tagged type, or a noninterface synchronized tagged type.]

Language Design Principles

1.a We want to make sure that we can extend a generic formal tagged type, without knowing its discriminants.

1.b We don't want to allow components in an extension aggregate to depend on discriminants inherited from the parent value, since such dependence requires staticness in aggregates, at least for variants.

Syntax

2 record_extension_part ::= **with** record_definition

Legality Rules

3/2 {*AI95-00344-01*} {*AI95-00345-01*} {*AI95-00419-01*} The parent type of a record extension shall not be a class-wide type nor shall it be a synchronized tagged type (see 3.9.4). If the parent type or any progenitor is nonlimited, then each of the components of the record_extension_part shall be nonlimited. In addition to the places where Legality Rules normally apply (see 12.3), these rules apply also in the private part of an instance of a generic unit.

3.a **Reason:** If the parent is a limited formal type, then the actual might be nonlimited.

3.b/2 {*AI95-00344-01*} Ada 95 required the record extensions to be the same level as the parent type. Now we use accessibility checks on class-wide allocators and return statements to prevent objects from living longer than their type.

3.c/2 {*AI95-00345-01*} Synchronized tagged types cannot be extended. We have this limitation so that all of the data of a task or protected type is defined within the type. Data defined outside of the type wouldn't be subject to the mutual exclusion properties of a protected type, and couldn't be used by a task, and thus doesn't seem to be worth the potential impact on implementations.

4/2 {*AI95-00344-01*} Within the body of a generic unit, or the body of any of its descendant library units, a tagged type shall not be declared as a descendant of a formal type declared within the formal part of the generic unit.

4.a **Reason:** This paragraph ensures that a dispatching call will never attempt to execute an inaccessible subprogram body.

4.a.1/2 {*AI95-00344-01*} The convoluted wording (“formal type declared within the formal part”) is necessary to include tagged types that are formal parameters of formal packages of the generic unit, as well as formal tagged and tagged formal derived types of the generic unit.

4.b/2 {*AI95-00344-01*} This rule is necessary in order to preserve the contract model.

4.c/2 {*AI05-0005-1*} {*AI95-00344-01*} If an ancestor is a formal of the generic unit , we have a problem because it might have an unknown number of subprograms that require overriding, as in the following example:

4.d/2
```
package P is
    type T is tagged null record;
    function F return T; -- Inherited versions will require overriding.
end P;
```

4.e
```
generic
    type TT is tagged private;
package Gp is
    type NT is abstract new TT with null record;
    procedure Q(X : in NT) is abstract;
end Gp;
```

4.f/2
```
package body Gp is
    type NT2 is new NT with null record; -- Illegal!
    procedure Q(X : in NT2) is begin null; end Q;
    -- Is this legal or not? Can't decide because
    -- we don't know whether TT had any functions that require
    -- overriding on extension.
end Gp;
```

```
package I is new Gp(TT => P.T);
```

4.g

I.NT is an abstract type with two abstract subprograms: F (inherited as abstract) and Q (explicitly declared as abstract). But the generic body doesn't know about F, so we don't know that it needs to be overridden to make a nonabstract extension of NT. Hence, we have to disallow this case.

4.h/2

Similarly, since the actual type for a formal tagged limited private type can be a nonlimited type, we would have a problem if a type extension of a limited private formal type could be declared in a generic body. Such an extension could have a task component, for example, and an object of that type could be passed to a dispatching operation of a nonlimited ancestor type. That operation could try to copy the object with the task component. That would be bad. So we disallow this as well.

4.h.1/2

If TT were declared as abstract, then we could have the same problem with abstract procedures.

4.i

We considered disallowing all tagged types in a generic body, for simplicity. We decided not to go that far, in order to avoid unnecessary restrictions.

4.j

We also considered trying make the accessibility level part of the contract; i.e. invent some way of saying (in the generic_declaration) "all instances of this generic unit will have the same accessibility level as the generic_declaration". Unfortunately, that doesn't solve the part of the problem having to do with abstract types.

4.k

This paragraph was deleted.

4.l/2

Ramification: {*AI95-00344*} This rule applies to types with ancestors (directly or indirectly) of formal interface types (see 12.5.5), formal tagged private types (see 12.5.1), and formal derived private types whose ancestor type is tagged (see 12.5.1).

4.m/2

Static Semantics

{*AI95-00391-01*} A record extension is a *null extension* if its declaration has no known_discriminant_part and its record_extension_part includes no component_declarations.

4.1/2

{*AI12-0191-1*} In the case where the (compile-time) view of an object *X* is of a tagged type T1 or T1'Class and the (run-time) tag of *X* is T2'Tag, only the components (if any) of *X* that are components of T1 (or that are discriminants which correspond to a discriminant of T1) are said to be *components of the nominal type* of *X*. Similarly, only parts (respectively, subcomponents) of T1 are parts (respectively, subcomponents) of the nominal type of *X*.

4.2/5

Ramification: {*AI12-0191-1*} For example, if T2 is an undiscriminated extension of T1 which declares a component named Comp, then *X*.Comp is not a component of the nominal type of *X*.

4.n/5

Discussion: {*AI12-0191-1*} For example, there is a Dynamic Semantics rule that finalization of an object includes finalization of its components (see 7.6.1). In the following case:

4.o/5

```
type T1 is tagged null record;
type T2 is new T1 with record
   Comp : Some_Controlled_Type;
end record;
function Func return T1'Class is (T2'(others => <>));
X : T1'Class := Func;
```

4.p/5

the rule that "every component of the object is finalized" (as opposed to something like "every component of the nominal type of the object is finalized") means that the finalization of X will include finalization of X.Comp. For another example, see the rule about accessibility checking of access discriminants of parts of function results in 6.5. In contrast, the rules in 7.3.2 explicitly state that type invariant checks are only performed for parts which are of the type-invariant bearing type and which are parts of the nominal type of the object (as opposed to for all parts, whether part of the nominal type or not, which are of the invariant-bearing type). Similarly, the rule in 13.13.2 governing which components of a composite value are read and written by the default implementations of Read and Write for a composite type states that only the components of the object which are components of the nominal type of the object are read or written.

4.q/5

Dynamic Semantics

The elaboration of a record_extension_part consists of the elaboration of the record_definition.

5

NOTE 1 The term "type extension" refers to a type as a whole. The term "extension part" refers to the piece of text that defines the additional components (if any) the type extension has relative to its specified ancestor type.

6

Discussion: We considered other terminology, such as "extended type". However, the terms "private extended type" and "record extended type" did not convey the proper meaning. Hence, we have chosen to uniformly use the term "extension" as the type resulting from extending a type, with "private extension" being one produced by privately

6.a

extending the type, and “record extension” being one produced by extending the type with an additional record-like set of components. Note also that the term “type extension” refers to the result of extending a type in the language Oberon as well (though there the term “extended type” is also used, interchangeably, perhaps because Oberon doesn't have the concept of a “private extension”).

7/2 NOTE 2 {*AI95-00344-01*} When an extension is declared immediately within a body, primitive subprograms are inherited and are overridable, but new primitive subprograms cannot be added.

8/5 NOTE 3 {*AI12-0447-1*} By the rules given in 3.8, a name that denotes a component (including a discriminant) of the parent type is illegal within the record_extension_part. Similarly, a name that denotes a component defined within the record_extension_part is illegal within the record_extension_part. A name that denotes a discriminant of the record extension is legal, providing that it refers to a discriminant defined in a new known_discriminant_part in the enclosing type declaration.

8.a **Reason:** The restriction against depending on discriminants of the parent is to simplify the definition of extension aggregates. The restriction against using parent components in other ways is methodological; it presumably simplifies implementation as well.

9/5 NOTE 4 {*AI12-0447-1*} By the rules given in 8.3, each visible component of a record extension will have a unique name, whether the component is (visibly) inherited from the parent type or declared in the record_extension_part.

Examples

10 *Examples of record extensions (of types defined above in 3.9):*

11
```
type Painted_Point is new Point with
  record
    Paint : Color := White;
  end record;
    -- Components X and Y are inherited
```

12
```
Origin : constant Painted_Point := (X | Y => 0.0, Paint => Black);
```

13
```
type Literal is new Expression with
  record                -- a leaf in an Expression tree
    Value : Real;
  end record;
```

14/5
```
{AI12-0404-1} type Expr_Ptr is access all Expression'Class;
                                -- see 3.9
```

15
```
type Binary_Operation is new Expression with
  record                -- an internal node in an Expression tree
    Left, Right : Expr_Ptr;
  end record;
```

16
```
type Addition is new Binary_Operation with null record;
type Subtraction is new Binary_Operation with null record;
  -- No additional components needed for these extensions
```

17
```
Tree : Expr_Ptr :=      -- A tree representation of "5.0 + (13.0–7.0)"
   new Addition'(
      Left  => new Literal'(Value => 5.0),
      Right => new Subtraction'(
         Left  => new Literal'(Value => 13.0),
         Right => new Literal'(Value => 7.0)));
```

Extensions to Ada 83

17.a Type extension is a new concept.

Extensions to Ada 95

17.b/2 {*AI95-00344-01*} Type extensions now can be declared in more nested scopes than their parent types. Additional accessibility checks on allocators and return statements prevent objects from outliving their type.

Wording Changes from Ada 95

17.c/2 {*AI95-00345-01*} Added wording to prevent extending synchronized tagged types.

17.d/2 {*AI95-00391-01*} Defined null extension for use elsewhere.

Wording Changes from Ada 2012

{*AI12-0191-1*} Defined the term "components of the nominal type" to remove a confusion as to how components are described in Dynamic Semantics rules. 17.e/5

3.9.2 Dispatching Operations of Tagged Types

{*AI95-00260-02*} {*AI95-00335-01*} {*AI12-0342-1*} {*AI12-0419-1*} The primitive subprograms of a tagged type, the subprograms declared by formal_abstract_subprogram_declarations, the Put_Image attribute (see 4.10) of a specific tagged type, and the stream attributes of a specific tagged type that are available (see 13.13.2) at the end of the declaration list where the type is declared are called *dispatching operations*. [A dispatching operation can be called using a statically determined *controlling* tag, in which case the body to be executed is determined at compile time. Alternatively, the controlling tag can be dynamically determined, in which case the call *dispatches* to a body that is determined at run time;] such a call is termed a *dispatching call*. [As explained below, the properties of the operands and the context of a particular call on a dispatching operation determine how the controlling tag is determined, and hence whether or not the call is a dispatching call. Run-time polymorphism is achieved when a dispatching operation is called by a dispatching call.] 1/5

Reason: {*AI95-00335-01*} For the stream attributes of a type declared immediately within a package_specification that has a partial view, the declaration list to consider is the visible part of the package. Stream attributes that are not available in the same declaration list are not dispatching as there is no guarantee that descendants of the type have available attributes (there is such a guarantee for visibly available attributes). If we allowed dispatching for any available attribute, then for attributes defined in the private part we could end up executing a nonexistent body. 1.a.1/2

Language Design Principles

The controlling tag determination rules are analogous to the overload resolution rules, except they deal with run-time type identification (tags) rather than compile-time type resolution. As with overload resolution, controlling tag determination may depend on operands or result context. 1.a

Static Semantics

{*AI95-00260-02*} {*AI95-00416-01*} {*AI05-0076-1*} A *call on a dispatching operation* is a call whose name or prefix denotes the declaration of a dispatching operation. A *controlling operand* in a call on a dispatching operation of a tagged type *T* is one whose corresponding formal parameter is of type *T* or is of an anonymous access type with designated type *T*; the corresponding formal parameter is called a *controlling formal parameter*. If the controlling formal parameter is an access parameter, the controlling operand is the object designated by the actual parameter, rather than the actual parameter itself. If the call is to a (primitive) function with result type *T* (a *function with a controlling result*), then the call has a *controlling result* — the context of the call can control the dispatching. Similarly, if the call is to a function with an access result type designating *T* (a *function with a controlling access result*), then the call has a *controlling access result*, and the context can similarly control dispatching. 2/3

Ramification: This definition implies that a call through the dereference of an access-to-subprogram value is never considered a call on a dispatching operation. Note also that if the prefix denotes a renaming_declaration, the place where the renaming occurs determines whether it is primitive; the thing being renamed is irrelevant. 2.a

{*AI12-0236-1*} A name or expression of a tagged type is either *statically* tagged, *dynamically* tagged, or *tag indeterminate*, according to whether, when used as a controlling operand, the tag that controls dispatching is determined statically by the operand's (specific) type, dynamically by its tag at run time, or from context. A qualified_expression or parenthesized expression is statically, dynamically, or indeterminately tagged according to its operand. A conditional_expression is statically, dynamically, or indeterminately tagged according to rules given in 4.5.7. A declare_expression is statically, dynamically, or indeterminately tagged according to its *body*_expression. For other kinds of names and expressions, this is determined as follows: 3/5

4/2 • {*AI95-00416-01*} The name or expression is *statically tagged* if it is of a specific tagged type and, if it is a call with a controlling result or controlling access result, it has at least one statically tagged controlling operand;

4.a **Discussion:** It is illegal to have both statically tagged and dynamically tagged controlling operands in the same call -- see below.

5/2 • {*AI95-00416-01*} The name or expression is *dynamically tagged* if it is of a class-wide type, or it is a call with a controlling result or controlling access result and at least one dynamically tagged controlling operand;

6/2 • {*AI95-00416-01*} The name or expression is *tag indeterminate* if it is a call with a controlling result or controlling access result, all of whose controlling operands (if any) are tag indeterminate.

7/1 {*8652/0010*} {*AI95-00127-01*} [A type_conversion is statically or dynamically tagged according to whether the type determined by the subtype_mark is specific or class-wide, respectively.] For an object that is designated by an expression whose expected type is an anonymous access-to-specific tagged type, the object is dynamically tagged if the expression, ignoring enclosing parentheses, is of the form X'Access, where X is of a class-wide type, or is of the form **new** T'(...), where T denotes a class-wide subtype. Otherwise, the object is statically or dynamically tagged according to whether the designated type of the type of the expression is specific or class-wide, respectively.

7.a **Ramification:** A type_conversion is never tag indeterminate, even if its operand is. A designated object is never tag indeterminate.

7.a.1/1 {*8652/0010*} {*AI95-00127-01*} Allocators and access attributes of class-wide types can be used as the controlling parameters of dispatching calls.

Legality Rules

8 A call on a dispatching operation shall not have both dynamically tagged and statically tagged controlling operands.

8.a **Reason:** This restriction is intended to minimize confusion between whether the dynamically tagged operands are implicitly converted to, or tag checked against the specific type of the statically tagged operand(s).

9/1 {*8652/0010*} {*AI95-00127-01*} If the expected type for an expression or name is some specific tagged type, then the expression or name shall not be dynamically tagged unless it is a controlling operand in a call on a dispatching operation. Similarly, if the expected type for an expression is an anonymous access-to-specific tagged type, then the object designated by the expression shall not be dynamically tagged unless it is a controlling operand in a call on a dispatching operation.

9.a **Reason:** This prevents implicit "truncation" of a dynamically-tagged value to the specific type of the target object/formal. An explicit conversion is required to request this truncation.

9.b/4 **Ramification:** {*AI95-00252-01*} {*AI12-0039-1*} This rule applies to all expressions or names with a specific expected type, not just those that are actual parameters to a dispatching call. This rule does not apply to a membership test whose *tested*_simple_expression is class-wide, since any type that covers the tested type is explicitly allowed. See 4.5.2. This rule also doesn't apply to a selected_component whose selector_name is a subprogram, since the rules explicitly say that the prefix may be class-wide (see 4.1.3).

10/2 {*8652/0011*} {*AI95-00117-01*} {*AI95-00430-01*} In the declaration of a dispatching operation of a tagged type, everywhere a subtype of the tagged type appears as a subtype of the profile (see 6.1), it shall statically match the first subtype of the tagged type. If the dispatching operation overrides an inherited subprogram, it shall be subtype conformant with the inherited subprogram. The convention of an inherited dispatching operation is the convention of the corresponding primitive operation of the parent or progenitor type. The default convention of a dispatching operation that overrides an inherited primitive operation is the convention of the inherited operation; if the operation overrides multiple inherited operations, then they shall all have the same convention. An explicitly declared dispatching operation shall not be of convention Intrinsic.

Reason: These rules ensure that constraint checks can be performed by the caller in a dispatching call, and parameter passing conventions match up properly. A special rule on aggregates prevents values of a tagged type from being created that are outside of its first subtype. 10.a

{*AI95-00416-01*} The default_expression for a controlling formal parameter of a dispatching operation shall be tag indeterminate. 11/2

Reason: {*AI95-00416-01*} This rule ensures that the default_expression always produces the "correct" tag when called with or without dispatching, or when inherited by a descendant. If it were statically tagged, the default would be useless for a dispatching call; if it were dynamically tagged, the default would be useless for a nondispatching call. 11.a/2

{*AI95-00404-01*} If a dispatching operation is defined by a subprogram_renaming_declaration or the instantiation of a generic subprogram, any access parameter of the renamed subprogram or the generic subprogram that corresponds to a controlling access parameter of the dispatching operation, shall have a subtype that excludes null. 11.1/2

A given subprogram shall not be a dispatching operation of two or more distinct tagged types. 12

Reason: This restriction minimizes confusion since multiple dispatching is not provided. The normal solution is to replace all but one of the tagged types with their class-wide types. 12.a

Ramification: {*8652/0098*} {*AI95-00183-01*} This restriction applies even if the partial view (see 7.3) of one or both of the types is untagged. This follows from the definition of dispatching operation: the operation is a dispatching operation anywhere the full views of the (tagged) types are visible. 12.a.1/1

The explicit declaration of a primitive subprogram of a tagged type shall occur before the type is frozen (see 13.14). [For example, new dispatching operations cannot be added after objects or values of the type exist, nor after deriving a record extension from it, nor after a body.] 13

Reason: {*AI95-00344-01*} This rule is needed because (1) we don't want people dispatching to things that haven't been declared yet, and (2) we want to allow the static part of tagged type descriptors to be static (allocated statically, and initialized to link-time-known symbols). Suppose T2 inherits primitive P from T1, and then overrides P. Suppose P is called *before* the declaration of the overriding P. What should it dispatch to? If the answer is the new P, we've violated the first principle above. If the answer is the old P, we've violated the second principle. (A call to the new one necessarily raises Program_Error, but that's beside the point.) 13.a/2

Note that a call upon a dispatching operation of type *T* will freeze *T*. 13.b

We considered applying this rule to all derived types, for uniformity. However, that would be upward incompatible, so we rejected the idea. As in Ada 83, for an untagged type, the above call upon P will call the old P (which is arguably confusing). 13.c

Implementation Note: {*AI95-00326-01*} Because of this rule, the type descriptor can be created (presumably containing linker symbols pointing at the not-yet-compiled bodies) at the first freezing point of the type. It also prevents, for a (nonincomplete) tagged type declared in a package_specification, overriding in the body or by a child subprogram. 13.d/2

Ramification: {*AI95-00251-01*} A consequence is that for a tagged type declaration in a declarative_part, only the last (overriding) primitive subprogram can be declared by a subprogram_body. (Other overridings must be provided by subprogram_declarations.) 13.e/2

To be honest: {*AI05-0222-1*} This rule applies only to "original" declarations and not to the completion of a primitive subprogram, even though a completion is technically an explicit declaration, and it may declare a primitive subprogram. 13.f/3

Dynamic Semantics

For the execution of a call on a dispatching operation of a type *T*, the *controlling tag value* determines which subprogram body is executed. The controlling tag value is defined as follows: 14

- If one or more controlling operands are statically tagged, then the controlling tag value is *statically determined* to be the tag of *T*. 15
- If one or more controlling operands are dynamically tagged, then the controlling tag value is not statically determined, but is rather determined by the tags of the controlling operands. If there is more than one dynamically tagged controlling operand, a check is made that they all have the same tag. If this check fails, Constraint_Error is raised unless the call is a function_call whose 16

name denotes the declaration of an equality operator (predefined or user defined) that returns Boolean, in which case the result of the call is defined to indicate inequality, and no subprogram_body is executed. This check is performed prior to evaluating any tag-indeterminate controlling operands.

16.a **Reason:** Tag mismatch is considered an error (except for "=" and "/=") since the corresponding primitive subprograms in each specific type expect all controlling operands to be of the same type. For tag mismatch with an equality operator, rather than raising an exception, "=" returns False and "/=" returns True. No equality operator is actually invoked, since there is no common tag value to control the dispatch. Equality is a special case to be consistent with the existing Ada 83 principle that equality comparisons, even between objects with different constraints, never raise Constraint_Error.

17/2 - {*AI95-00196-01*} If all of the controlling operands (if any) are tag-indeterminate, then:

18/2 - {*AI95-00239-01*} {*AI95-00416-01*} If the call has a controlling result or controlling access result and is itself, or designates, a (possibly parenthesized or qualified) controlling operand of an enclosing call on a dispatching operation of a descendant of type *T*, then its controlling tag value is determined by the controlling tag value of this enclosing call;

18.a/2 **Discussion:** {*AI95-00239-01*} For code that a user can write explicitly, the only contexts that can control dispatching of a function with a controlling result of type T are those that involve controlling operands of the same type T: if the two types differ there is an illegality and the dynamic semantics are irrelevant.

18.b/2 In the case of an inherited subprogram however, if a default expression is a function call, it may be of type T while the parameter is of a type derived from T. To cover this case, we talk about "a descendant of T" above. This is safe, because if the type of the parameter is descended from the type of the function result, it is guaranteed to inherit or override the function, and this ensures that there will be an appropriate body to dispatch to. Note that abstract functions are not an issue here because the call to the function is a dispatching call, so it is guaranteed to always land on a concrete body.

18.1/2 - {*AI95-00196-01*} {*AI95-00416-01*} If the call has a controlling result or controlling access result and (possibly parenthesized, qualified, or dereferenced) is the expression of an assignment_statement whose target is of a class-wide type, then its controlling tag value is determined by the target;

19 - Otherwise, the controlling tag value is statically determined to be the tag of type *T*.

19.a **Ramification:** This includes the cases of a tag-indeterminate procedure call, and a tag-indeterminate function_call that is used to initialize a class-wide formal parameter or class-wide object.

20/3 {*AI95-00345-01*} {*AI05-0126-1*} For the execution of a call on a dispatching operation, the action performed is determined by the properties of the corresponding dispatching operation of the specific type identified by the controlling tag value:

20.1/3 - {*AI05-0126-1*} if the corresponding operation is explicitly declared for this type, [even if the declaration occurs in a private part], then the action comprises an invocation of the explicit body for the operation;

20.2/3 - {*AI95-00345-01*} {*AI05-0126-1*} if the corresponding operation is implicitly declared for this type and is implemented by an entry or protected subprogram (see 9.1 and 9.4), then the action comprises a call on this entry or protected subprogram, with the target object being given by the first actual parameter of the call, and the actual parameters of the entry or protected subprogram being given by the remaining actual parameters of the call, if any;

20.3/3 - {*AI05-0197-1*} if the corresponding operation is a predefined operator then the action comprises an invocation of that operator;

20.4/3 - {*AI95-00345-01*} {*AI05-0126-1*} {*AI05-0197-1*} {*AI05-0250-1*} {*AI05-0254-1*} otherwise, the action is the same as the action for the corresponding operation of the parent type or progenitor type from which the operation was inherited except that additional invariant checks (see 7.3.2) and class-wide postcondition checks (see 6.1.1) may apply. If there is more than one such corresponding operation, the action is that for the operation that is not a null procedure, if any; otherwise, the action is that of an arbitrary one of the operations.

This paragraph was deleted.{*AI05-0126-1*} 20.a/3

Ramification: {*AI05-0005-1*} {*AI05-0126-1*} "Corresponding dispatching operation" refers to the inheritance relationship between subprograms. Primitive operations are always inherited for a type T, but they might not be declared if the primitive operation is never visible within the immediate scope of the type T. If no corresponding operation is declared, the last bullet is used and the corresponding operation of the parent type is executed (an explicit body that happens to have the same name and profile is not called in that case). 20.a.1/3

{*AI05-0005-1*} {*AI05-0126-1*} We have to talk about progenitors in the last bullet in case the corresponding operation is a null procedure inherited from an interface. In that case, the parent type might not even have the operation in question. 20.a.2/3

{*AI05-0197-1*} For the last bullet, if there are multiple corresponding operations for the parent and progenitors, all but one of them have to be a null procedure. (If the progenitors declared abstract routines, there would have to be an explicit overriding of the operation, and then the first bullet would apply.) We call the nonnull routine if one exists. 20.a.3/3

{*AI05-0126-1*} Any explicit declaration for an inherited corresponding operation has to be an overriding routine. These rules mean that a dispatching call executes the overriding routine (if any) for the specific type. 20.a.4/3

Reason: {*AI05-0005-1*} The wording of the above rules is intended to ensure that the same body is executed for a given tag, whether that tag is determined statically or dynamically. For a type declared in a package, it doesn't matter whether a given subprogram is overridden in the visible part or the private part, and it doesn't matter whether the call is inside or outside the package. For example: 20.b/3

```
package P1 is                                                                    20.c
    type T1 is tagged null record;
    procedure Op_A(Arg : in T1);
    procedure Op_B(Arg : in T1);
end P1;

with P1; use P1;                                                                 20.d
package P2 is
    type T2 is new T1 with null record;
    procedure Op_A(Param : in T2);
private
    procedure Op_B(Param : in T2);
end P2;

with P1; with P2;                                                                20.e/1
procedure Main is
    X : P2.T2;
    Y : P1.T1'Class := X;
begin
    P2.Op_A(Param => X); -- Nondispatching call to a dispatching operation.
    P1.Op_A(Arg => Y); -- Dispatching call.
    P2.Op_B(Arg => X); -- Nondispatching call to a dispatching operation.
    P1.Op_B(Arg => Y); -- Dispatching call.
end Main;
```

The two calls to Op_A both execute the body of Op_A that has to occur in the body of package P2. Similarly, the two calls to Op_B both execute the body of Op_B that has to occur in the body of package P2, even though Op_B is overridden in the private part of P2. Note, however, that the formal parameter names are different for P2.Op_A versus P2.Op_B. The overriding declaration for P2.Op_B is not visible in Main, so the name in the call actually denotes the implicit declaration of Op_B inherited from T1. 20.f

If a call occurs in the program text before an overriding, which can happen only if the call is part of a default expression, the overriding will still take effect for that call. 20.g

Implementation Note: Even when a tag is not *statically determined*, a compiler might still be able to figure it out and thereby avoid the overhead of run-time dispatching. 20.h

NOTE 1 The body to be executed for a call on a dispatching operation is determined by the tag; it does not matter whether that tag is determined statically or dynamically, and it does not matter whether the subprogram's declaration is visible at the place of the call. 21

NOTE 2 {*AI95-00260-02*} This subclause covers calls on dispatching subprograms of a tagged type. Rules for tagged type membership tests are described in 4.5.2. Controlling tag determination for an assignment_statement is described in 5.2. 22/2

NOTE 3 A dispatching call can dispatch to a body whose declaration is not visible at the place of the call. 23

NOTE 4 A call through an access-to-subprogram value is never a dispatching call, even if the access value designates a dispatching operation. Similarly a call whose prefix denotes a subprogram_renaming_declaration cannot be a dispatching call unless the renaming itself is the declaration of a primitive subprogram. 24

Extensions to Ada 83

24.a The concept of dispatching operations is new.

Incompatibilities With Ada 95

24.b/2 {*AI95-00404-01*} If a dispatching operation is defined by a subprogram_renaming_declaration, and it has a controlling access parameter, Ada 2005 requires the subtype of the parameter to exclude null. The same applies to instantiations. This is required so that all calls to the subprogram operate the same way (controlling access parameters have to exclude null so that dispatching calls will work). Since Ada 95 didn't have the notion of access subtypes that exclude null, and all access parameters excluded null, it had no such rules. These rules will require the addition of an explicit **not null** on nondispatching operations that are later renamed to be dispatching, or on a generic that is used to define a dispatching operation.

Extensions to Ada 95

24.c/2 {*AI95-00416-01*} Functions that have an access result type can be dispatching in the same way as a function that returns a tagged object directly.

Wording Changes from Ada 95

24.d/3 {*8652/0010*} {*AI95-00127-01*} {*AI05-0299-1*} **Corrigendum:** Allocators and access attributes of objects of class-wide types can be used as the controlling parameter in a dispatching calls. This was an oversight in the definition of Ada 95. (See 3.10.2 and 4.8).

24.e/2 {*8652/0011*} {*AI95-00117-01*} {*AI95-00430-01*} **Corrigendum:** Corrected the conventions of dispatching operations. This is extended in Ada 2005 to cover operations inherited from progenitors, and to ensure that the conventions of all inherited operations are the same.

24.f/2 {*AI95-00196-01*} Clarified the wording to ensure that functions with no controlling operands are tag-indeterminate, and to describe that the controlling tag can come from the target of an assignment_statement.

24.g/2 {*AI95-00239-01*} Fixed the wording to cover default expressions inherited by derived subprograms. A literal reading of the old wording would have implied that operations would be called with objects of the wrong type.

24.h/2 {*AI95-00260-02*} An abstract formal subprogram is a dispatching operation, even though it is not a primitive operation. See 12.6, “Formal Subprograms”.

24.i/2 {*AI95-00345-01*} Dispatching calls include operations implemented by entries and protected operations, so we have to update the wording to reflect that.

24.j/2 {*AI95-00335-01*} A stream attribute of a tagged type is usually a dispatching operation, even though it is not a primitive operation. If they weren't dispatching, T'Class'Input and T'Class'Output wouldn't work.

Wording Changes from Ada 2005

24.k/3 {*AI05-0076-1*} **Correction:** Defined “function with a controlling result”, as it is used in 3.9.3.

24.l/3 {*AI05-0126-1*} {*AI05-0197-1*} **Correction:** Corrected holes in the definition of dynamic dispatching: the behavior for operations that are never declared and/or inherited from a progenitor were not specified.

Wording Changes from Ada 2012

24.m/5 {*AI12-0236-1*} Added a special rule for taggedness of declare_expressions, and added a pointer to the existing special rules for conditional_expressions, as the list of exceptions to the usual rules appear to be exhaustive.

24.n/5 {*AI12-0419-1*} Added wording to clarify that the Put_Image attribute is a dispatching operation. Put_Image follows the model of stream-oriented attributes, and thus need to be mentioned in the same place.

3.9.3 Abstract Types and Subprograms

1/2 {*AI95-00345-01*} [An *abstract type* is a tagged type intended for use as an ancestor of other types, but which is not allowed to have objects of its own. An *abstract subprogram* is a subprogram that has no body, but is intended to be overridden at some point when inherited. Because objects of an abstract type cannot be created, a dispatching call to an abstract subprogram always dispatches to some overriding body.]

1.a.1/5 **Term entry: abstract type** — tagged type intended for use as an ancestor of other types, but which is not allowed to have objects of its own

Language Design Principles

{*AI05-0299-1*} An abstract subprogram has no body, so the rules in this subclause are designed to ensure (at compile time) that the body will never be invoked. We do so primarily by disallowing the creation of values of the abstract type. Therefore, since type conversion and parameter passing don't change the tag, we know we will never get a class-wide value with a tag identifying an abstract type. This means that we only have to disallow nondispatching calls on abstract subprograms (dispatching calls will never reach them). 1.a/3

Syntax

{*AI95-00218-03*} {*AI95-00348-01*} {*AI05-0183-1*} abstract_subprogram_declaration ::= 1.1/3
[overriding_indicator]
subprogram_specification **is abstract**
[aspect_specification];

Static Semantics

{*AI95-00345-01*} Interface types (see 3.9.4) are abstract types. In addition, a tagged type that has the reserved word **abstract** in its declaration is an abstract type. The class-wide type (see 3.4.1) rooted at an abstract type is not itself an abstract type. 1.2/2

Legality Rules

{*AI95-00345-01*} Only a tagged type shall have the reserved word **abstract** in its declaration. 2/2

Ramification: Untagged types are never abstract, even though they can have primitive abstract subprograms. Such subprograms cannot be called, unless they also happen to be dispatching operations of some tagged type, and then only via a dispatching call. 2.a

Class-wide types are never abstract. If T is abstract, then it is illegal to declare a stand-alone object of type T, but it is OK to declare a stand-alone object of type T'Class; the latter will get a tag from its initial value, and this tag will necessarily be different from T'Tag. 2.b

{*AI95-00260-02*} {*AI95-00348-01*} A subprogram declared by an abstract_subprogram_declaration or a formal_abstract_subprogram_declaration (see 12.6) is an *abstract subprogram*. If it is a primitive subprogram of a tagged type, then the tagged type shall be abstract. 3/2

Ramification: Note that for a private type, this applies to both views. The following is illegal: 3.a

```
package P is                                                          3.b
    type T is abstract tagged private;
    function Foo (X : T) return Boolean is abstract; -- Illegal!
private
    type T is tagged null record; -- Illegal!
    X : T;
    Y : Boolean := Foo (T'Class (X));
end P;
```

The full view of T is not abstract, but has an abstract operation Foo, which is illegal. The two lines marked "-- *Illegal!*" are illegal when taken together. 3.c

Reason: {*AI95-00310-01*} We considered disallowing untagged types from having abstract primitive subprograms. However, we rejected that plan, because it introduced some silly anomalies, and because such subprograms are harmless. For example: 3.d/2

```
package P is                                                          3.e/1
   type Field_Size is range 0..100;
   type T is abstract tagged null record;
   procedure Print(X : in T; F : in Field_Size := 0) is abstract;
  . . .
package Q is
   type My_Field_Size is new Field_Size;
   -- implicit declaration of Print(X : T; F : My_Field_Size := 0) is abstract;
end Q;
```

It seemed silly to make the derivative of My_Field_Size illegal, just because there was an implicitly declared abstract subprogram that was not primitive on some tagged type. Other rules could be formulated to solve this problem, but the current ones seem like the simplest. 3.f

3.g/2 {*AI95-00310-01*} In Ada 2005, abstract primitive subprograms of an untagged type may be used to "undefine" an operation.

3.h/2 **Ramification:** {*AI95-00260-02*} Note that the second sentence does not apply to abstract formal subprograms, as they are never primitive operations of a type.

4/3 {*AI95-00251-01*} {*AI95-00334-01*} {*AI95-00391-01*} {*AI05-0097-1*} {*AI05-0198-1*} If a type has an implicitly declared primitive subprogram that is inherited or is a predefined operator, and the corresponding primitive subprogram of the parent or ancestor type is abstract or is a function with a controlling access result, or if a type other than a nonabstract null extension inherits a function with a controlling result, then:

4.a/3 **Ramification:** {*AI05-0068-1*} These rules apply to each view of the type individually. That is necessary to preserve privacy. For instance, in the following example:

4.b/3
```
package P is
   type I is interface;
   procedure Op (X : I) is abstract;
end P;
```

4.c/3
```
with P;
package Q is
   type T is abstract new P.I with private;
   -- Op inherited here.
private
   type T is abstract new P.I with null record;
   procedure Op (X : T) is null;
end Q;
```

4.d/3
```
with Q;
package R is
   type T2 is new Q.T with null record;
   -- Illegal. Op inherited here, but requires overriding.
end R;
```

4.e/3 If this did not depend on the view, this would be legal. But in that case, the fact that Op is overridden in the private part would be visible; package R would have to be illegal if no overriding was in the private part.

4.f/3 Note that this means that whether an inherited subprogram is abstract or concrete depends on where it inherited. In the case of Q, Q.Op in the visible part is abstract, while Q.Op in the private part is concrete. That is, R is illegal since it is an unrelated unit (and thus it cannot see the private part), but if R had been a private child of Q, it would have been legal.

5/2 - {*AI95-00251-01*} {*AI95-00334-01*} If the type is abstract or untagged, the implicitly declared subprogram is *abstract*.

5.a **Ramification:** Note that it is possible to override a concrete subprogram with an abstract one.

6/5 - {*AI95-00391-01*} {*AI12-0080-1*} {*AI12-0444-1*} Otherwise, the subprogram shall be overridden with a nonabstract subprogram or, in the case of a private extension inheriting a nonabstract function with a controlling result, have a full type that is a null extension[; for a type declared in the visible part of a package, the overriding may be either in the visible or the private part]. Such a subprogram is said to *require overriding*. However, if the type is a generic formal type, the subprogram is allowed to be inherited as is, without being overridden for the formal type itself; [a nonabstract version will necessarily be provided by the actual type.]

6.a/2 **Reason:** {*AI95-00228-01*} {*AI95-00391-01*} A function that returns the parent type requires overriding for a type extension (or becomes abstract for an abstract type) because conversion from a parent type to a type extension is not defined, and function return semantics is defined in terms of conversion (other than for a null extension; see below). (Note that parameters of mode **in out** or **out** do not have this problem, because the tag of the actual is not changed.)

6.b Note that the overriding required above can be in the private part, which allows the following:

6.c
```
package Pack1 is
    type Ancestor is abstract ...;
    procedure Do_Something(X : in Ancestor) is abstract;
end Pack1;
```

```
with Pack1; use Pack1;                                                    6.d
package Pack2 is
    type T1 is new Ancestor with record ...;
        -- A concrete type.
    procedure Do_Something(X : in T1); -- Have to override.
end Pack2;
```

```
with Pack1; use Pack1;                                                    6.e
with Pack2; use Pack2;
package Pack3 is
    type T2 is new Ancestor with private;
        -- A concrete type.
private
    type T2 is new T1 with -- Parent different from ancestor.
      record ... end record;
    -- Here, we inherit Pack2.Do_Something.
end Pack3;
```

{*AI95-00228-01*} T2 inherits an abstract Do_Something, but T2 is not abstract, so Do_Something has to be overridden. However, it is OK to override it in the private part. In this case, we override it by inheriting a concrete version from a different type. Nondispatching calls to Pack3.Do_Something are allowed both inside and outside package Pack3, as the client "knows" that the subprogram was necessarily overridden somewhere. 6.f/2

{*AI95-00391-01*} For a null extension, the result of a function with a controlling result is defined in terms of an extension_aggregate with a **null record** extension part (see 3.4). This means that these restrictions on functions with a controlling result do not have to apply to null extensions. 6.g/2

{*AI95-00391-01*} However, functions with controlling access results still require overriding. Changing the tag in place might clobber a preexisting object, and allocating new memory would possibly change the pool of the object, leading to storage leaks. Moreover, copying the object isn't possible for limited types. We don't need to restrict functions that have an access return type of an untagged type, as derived types with primitive subprograms have to have the same representation as their parent type. 6.h/2

{*AI12-0413-1*} A call on an abstract subprogram shall be a dispatching call; nondispatching calls to an abstract subprogram are not allowed. In addition to the places where Legality Rules normally apply (see 12.3), these rules also apply in the private part of an instance of a generic unit. 7/5

Ramification: {*AI95-00310-01*} {*AI12-0413-1*} If an abstract subprogram is not a dispatching operation of some tagged type, then it cannot be called at all. In Ada 2005, such subprograms are not even considered by name resolution (see 6.4). However, this rule is still needed for cases that can arise in the instance of a generic specification where Name Resolution Rules are not reapplied, but Legality Rules are, when the equality operator for an untagged record type is abstract, while the operator for the formal type is not abstract (see 12.5). 7.a/5

{*AI12-0189-1*} {*AI12-0292-1*} {*AI12-0320-1*} If the name or prefix given in an iterator_procedure_call (see 5.5.3) denotes an abstract subprogram, the subprogram shall be a dispatching subprogram. 7.1/5

{*AI05-0073-1*} {*AI05-0203-1*} {*AI12-0437-1*} The type of an aggregate, of an object created by an object_declaration or an allocator, or of a generic formal object of mode **in**, shall not be abstract. The type of the target of an assignment operation (see 5.2) shall not be abstract. The type of a component shall not be abstract. If the result type of a function is abstract, then the function shall be abstract. If a function has an access result type designating an abstract type, then the function shall be abstract. The type denoted by a return_subtype_indication (see 6.5) shall not be abstract. A generic function shall not have an abstract result type or an access result type designating an abstract type. 8/5

Reason: This ensures that values of an abstract type cannot be created, which ensures that a dispatching call to an abstract subprogram will not try to execute the nonexistent body. 8.a

Generic formal objects of mode **in** are like constants; therefore they should be forbidden for abstract types. Generic formal objects of mode **in out** are like renamings; therefore, abstract types are OK for them, though probably not terribly useful. 8.b

{*AI05-0073-1*} Generic functions returning a formal abstract type are illegal because any instance would have to be instantiated with a nonabstract type in order to avoid violating the function rule (generic functions cannot be declared abstract). But that would be an implied contract; it would be better for the contract to be explicit by the formal type not being declared abstract. Moreover, the implied contract does not add any capability. 8.c/3

9 If a partial view is not abstract, the corresponding full view shall not be abstract. If a generic formal type is abstract, then for each primitive subprogram of the formal that is not abstract, the corresponding primitive subprogram of the actual shall not be abstract.

9.a **Discussion:** By contrast, we allow the actual type to be nonabstract even if the formal type is declared abstract. Hence, the most general formal tagged type possible is "**type** T(<>) **is abstract tagged limited private**;".

9.b For an abstract private extension declared in the visible part of a package, it is only possible for the full type to be nonabstract if the private extension has no abstract dispatching operations.

9.c/2 **To be honest:** {*AI95-00294-01*} In the sentence about primitive subprograms above, there is some ambiguity as to what is meant by "corresponding" in the case where an inherited operation is overridden. This is best explained by an example, where the implicit declarations are shown as comments:

9.d/2
```
package P1 is
   type T1 is abstract tagged null record;
   procedure P (X : T1); -- (1)
end P1;
```

9.e/2
```
package P2 is
   type T2 is abstract new P1.T1 with null record;
   -- procedure P (X : T2); -- (2)
   procedure P (X : T2) is abstract; -- (3)
end P2;
```

9.f/2
```
generic
   type D is abstract new P1.T1 with private;
   -- procedure P (X : D); -- (4)
procedure G (X : D);
```

9.g/2
```
procedure I is new G (P2.T2); -- Illegal.
```

9.h/2 Type T2 inherits a nonabstract procedure P (2) from the primitive procedure P (1) of T1. P (2) is overridden by the explicitly declared abstract procedure P (3). Type D inherits a nonabstract procedure P (4) from P (1). In instantiation I, the operation corresponding to P (4) is the one which is not overridden, that is, P (3): the overridden operation P (2) does not "reemerge". Therefore, the instantiation is illegal.

10/3 {*AI05-0073-1*} For an abstract type declared in a visible part, an abstract primitive subprogram shall not be declared in the private part, unless it is overriding an abstract subprogram implicitly declared in the visible part. For a tagged type declared in a visible part, a primitive function with a controlling result or a controlling access result shall not be declared in the private part, unless it is overriding a function implicitly declared in the visible part.

10.a **Reason:** The "visible part" could be that of a package or a generic package. This rule is needed because a nonabstract type extension declared outside the package would not know about any abstract primitive subprograms or primitive functions with controlling results declared in the private part, and wouldn't know that they need to be overridden with nonabstract subprograms. The rule applies to a tagged record type or record extension declared in a visible part, just as to a tagged private type or private extension. The rule applies to explicitly and implicitly declared abstract subprograms:

10.b
```
package Pack is
    type T is abstract new T1 with private;
private
    type T is abstract new T2 with record ... end record;
    ...
end Pack;
```

10.c The above example would be illegal if T1 has a nonabstract primitive procedure P, but T2 overrides P with an abstract one; the private part should override P with a nonabstract version. On the other hand, if the P were abstract for both T1 and T2, the example would be legal as is.

11/2 {*AI95-00260-02*} A generic actual subprogram shall not be an abstract subprogram unless the generic formal subprogram is declared by a formal_abstract_subprogram_declaration. The prefix of an attribute_reference for the Access, Unchecked_Access, or Address attributes shall not denote an abstract subprogram.

11.a **Ramification:** An abstract_subprogram_declaration is not syntactically a subprogram_declaration. Nonetheless, an abstract subprogram is a subprogram, and an abstract_subprogram_declaration is a declaration of a subprogram.

11.b/2 {*AI95-00260-02*} The part about generic actual subprograms includes those given by default. Of course, an abstract formal subprogram's actual subprogram can be abstract.

Dynamic Semantics

{*AI95-00348-01*} The elaboration of an abstract_subprogram_declaration has no effect. 11.1/2

NOTE 1 {*AI12-0447-1*} Abstractness is not inherited; a type is abstract only if the reserved word **abstract** is used in the declaration of the type extension. 12/5

Ramification: A derived type can be abstract even if its parent is not. Similarly, an inherited concrete subprogram can be overridden with an abstract subprogram. 12.a

NOTE 2 A class-wide type is never abstract. Even if a class is rooted at an abstract type, the class-wide type for the class is not abstract, and an object of the class-wide type can be created; the tag of such an object will identify some nonabstract type in the class. 13

Examples

Example of an abstract type representing a set of natural numbers: 14

```
package Sets is                                                                    15
    subtype Element_Type is Natural;
    type Set is abstract tagged null record;
    function Empty return Set is abstract;
    function Union(Left, Right : Set) return Set is abstract;
    function Intersection(Left, Right : Set) return Set is abstract;
    function Unit_Set(Element : Element_Type) return Set is abstract;
    procedure Take(Element : out Element_Type;
                   From : in out Set) is abstract;
end Sets;
```

{*AI12-0442-1*} {*AI12-0452-1*} Given the above abstract type, one can derive various (nonabstract) extensions of the type, representing alternative implementations of a set. One possibility is to use a bit vector, but impose an upper bound on the largest element representable, while another possible implementation is a hash table, trading off space for flexibility. 16/5

Discussion: One way to export a type from a package with some components visible and some components private is as follows: 17.a

```
package P is                                                                       17.b
    type Public_Part is abstract tagged
        record
            ...
        end record;
    type T is new Public_Part with private;
    ...
private
    type T is new Public_Part with
        record
            ...
        end record;
end P;
```

The fact that Public_Part is abstract tells clients they have to create objects of type T instead of Public_Part. Note that the public part has to come first; it would be illegal to declare a private type Private_Part, and then a record extension T of it, unless T were in the private part after the full declaration of Private_Part, but then clients of the package would not have visibility to T. 17.c

Extensions to Ada 95

{*AI95-00391-01*} It is not necessary to override functions with a controlling result for a null extension. This makes it easier to derive a tagged type to complete a private type. 17.d/2

Wording Changes from Ada 95

{*AI95-00251-01*} {*AI95-00345-01*} Updated the wording to reflect the addition of interface types (see 3.9.4). 17.e/2

{*AI95-00260-02*} Updated the wording to reflect the addition of abstract formal subprograms (see 12.6). 17.f/2

{*AI95-00334-01*} The wording of shall-be-overridden was clarified so that it clearly applies to abstract predefined equality. 17.g/2

{*AI95-00348-01*} Moved the syntax and elaboration rule for abstract_subprogram_declaration here, so the syntax and most of the semantics are together (which is consistent with null procedures). 17.h/2

17.i/2 {*AI95-00391-01*} We define the term *require overriding* to make other wording easier to understand.

Incompatibilities With Ada 2005

17.j/3 {*AI05-0073-1*} **Correction:** Added rules to eliminate holes with controlling access results and generic functions that return abstract types. While these changes are technically incompatible, it is unlikely that they could be used in a program without violating some other rule of the use of abstract types.

17.k/3 {*AI05-0097-1*} **Correction:** Corrected a minor glitch having to do with abstract null extensions. The Ada 2005 rule allowed such extensions to inherit concrete operations in some rare cases. It is unlikely that these cases exist in user code.

Extensions to Ada 2005

17.l/3 {*AI05-0183-1*} An optional aspect_specification can be used in an abstract_subprogram_declaration. This is described in 13.1.1.

Wording Changes from Ada 2005

17.m/3 {*AI05-0198-1*} **Correction:** Clarified that the predefined operator corresponding to an inherited abstract operator is also abstract. The Ada 2005 rules caused the predefined operator and the inherited operator to override each other, which is weird. But the effect is the same either way (the operator is not considered for resolution).

17.n/3 {*AI05-0203-1*} **Correction:** Added wording to disallow abstract return objects. These were illegal in Ada 2005 by other rules; the extension to support class-wide type better opened a hole which has now been plugged.

Incompatibilities With Ada 2012

17.o/5 {*AI12-0413-1*} **Correction:** Clarified that a recheck is needed in the case of an actual that is a record type with an abstract equality. This is an incompatibility as the generic boilerplate was previously omitted, meaning that such a recheck should not have been performed in the private part of an instance. Usually, this would just change an elaboration time raise of Program_Error into an error (a good thing, as the instance will never be useful), but could break a working instance if the equality usage is in a default expression that appears in the private part of the generic unit and it is never used in a call. In that case, Ada 2022 will reject the instance while it would have worked in Ada 2012. As a practical matter, it's more likely that a compiler already does the recheck in the entire instance spec, or does not do it at all; thus for many implementations there will be no practical incompatibility.

3.9.4 Interface Types

1/2 {*AI95-00251-01*} {*AI95-00345-01*} [An interface type is an abstract tagged type that provides a restricted form of multiple inheritance. A tagged type, task type, or protected type may have one or more interface types as ancestors.]

1.a/5 **Term entry: interface type** — abstract tagged type that has no components or concrete operations except possibly null procedures
Note: Interface types are used for composing other interfaces and tagged types and thereby provide multiple inheritance. Only an interface type can be used as a progenitor of another type.

Language Design Principles

1.b/2 {*AI95-00251-01*} {*AI95-00345-01*} The rules are designed so that an interface can be used as either a parent type or a progenitor type without changing the meaning. That's important so that the order that interfaces are specified in a derived_type_definition is not significant. In particular, we want:

1.c/2
```
type Con1 is new Int1 and Int2 with null record;
type Con2 is new Int2 and Int1 with null record;
```

1.d/2 to mean exactly the same thing.

Syntax

2/2 {*AI95-00251-01*} {*AI95-00345-01*} interface_type_definition ::=
[**limited** | **task** | **protected** | **synchronized**] **interface** [**and** interface_list]

3/2 {*AI95-00251-01*} {*AI95-00419-01*} interface_list ::=
*interface*_subtype_mark {**and** *interface*_subtype_mark}

Static Semantics

{*AI95-00251-01*} An interface type (also called an *interface*) is a specific abstract tagged type that is defined by an interface_type_definition. 4/2

{*AI95-00345-01*} An interface with the reserved word **limited**, **task**, **protected**, or **synchronized** in its definition is termed, respectively, a *limited interface*, a *task interface*, a *protected interface*, or a *synchronized interface*. In addition, all task and protected interfaces are synchronized interfaces, and all synchronized interfaces are limited interfaces. 5/2

Term entry: synchronized — can be safely operated on by multiple tasks concurrently 5.a/5
Note: Synchronized is used to qualify entities, as in a synchronized interface.

{*AI95-00345-01*} {*AI95-00443-01*} [A task or protected type derived from an interface is a tagged type.] Such a tagged type is called a *synchronized* tagged type, as are synchronized interfaces and private extensions whose declaration includes the reserved word **synchronized**. 6/2

Proof: The full definition of tagged types given in 3.9 includes task and protected types derived from interfaces. 6.a/2

Ramification: The class-wide type associated with a tagged task type (including a task interface type) is a task type, because "task" is one of the language-defined classes of types (see 3.2). However, the class-wide type associated with an interface is *not* an interface type, as "interface" is *not* one of the language-defined classes (as it is not closed under derivation). In this sense, "interface" is similar to "abstract". The class-wide type associated with an interface is a concrete (nonabstract) indefinite tagged composite type. 6.b/2

"Private extension" includes generic formal private extensions, as explained in 12.5.1. 6.c/2

{*AI95-00345-01*} A task interface is an [abstract] task type. A protected interface is an [abstract] protected type. 7/2

Proof: The "abstract" follows from the definition of an interface type. 7.a/2

Reason: This ensures that task operations (like abort and the Terminated attribute) can be applied to a task interface type and the associated class-wide type. While there are no protected type operations, we apply the same rule to protected interfaces for consistency. 7.b/2

{*AI95-00251-01*} [An interface type has no components.] 8/2

Proof: This follows from the syntax and the fact that discriminants are not allowed for interface types. 8.a/2

{*AI95-00419-01*} An *interface*_subtype_mark in an interface_list names a *progenitor subtype*; its type is the *progenitor type*. An interface type inherits user-defined primitive subprograms from each progenitor type in the same way that a derived type inherits user-defined primitive subprograms from its progenitor types (see 3.4). 9/2

Term entry: progenitor — type given in the interface list, if any, of an interface, task, protected, or derived type definition 9.a/5
Note: A progenitor is always an interface type.

Legality Rules

{*AI95-00251-01*} All user-defined primitive subprograms of an interface type shall be abstract subprograms or null procedures. 10/2

{*AI95-00251-01*} The type of a subtype named in an interface_list shall be an interface type. 11/2

{*AI95-00251-01*} {*AI95-00345-01*} A type derived from a nonlimited interface shall be nonlimited. 12/2

{*AI95-00345-01*} An interface derived from a task interface shall include the reserved word **task** in its definition; any other type derived from a task interface shall be a private extension or a task type declared by a task declaration (see 9.1). 13/2

{*AI95-00345-01*} An interface derived from a protected interface shall include the reserved word **protected** in its definition; any other type derived from a protected interface shall be a private extension or a protected type declared by a protected declaration (see 9.4). 14/2

15/2 {*AI95-00345-01*} An interface derived from a synchronized interface shall include one of the reserved words **task**, **protected**, or **synchronized** in its definition; any other type derived from a synchronized interface shall be a private extension, a task type declared by a task declaration, or a protected type declared by a protected declaration.

15.a/2 **Reason:** We require that an interface descendant of a task, protected, or synchronized interface repeat the explicit kind of interface it will be, rather than simply inheriting it, so that a reader is always aware of whether the interface provides synchronization and whether it may be implemented only by a task or protected type. The only place where inheritance of the kind of interface might be useful would be in a generic if you didn't know the kind of the actual interface. However, the value of that is low because you cannot implement an interface properly if you don't know whether it is a task, protected, or synchronized interface. Hence, we require the kind of the actual interface to match the kind of the formal interface (see 12.5.5).

16/2 {*AI95-00345-01*} No type shall be derived from both a task interface and a protected interface.

16.a **Reason:** This prevents a single private extension from inheriting from both a task and a protected interface. For a private type, there can be no legal completion. For a generic formal derived type, there can be no possible matching type (so no instantiation could be legal). This rule provides early detection of the errors.

17/2 {*AI95-00251-01*} In addition to the places where Legality Rules normally apply (see 12.3), these rules apply also in the private part of an instance of a generic unit.

17.a/3 **Ramification:** {*AI05-0299-1*} This paragraph is intended to apply to all of the Legality Rules in this subclause. We cannot allow interface types which do not obey these rules, anywhere. Luckily, deriving from a formal type (which might be an interface) is not allowed for any tagged types in a generic body. So checking in the private part of a generic covers all of the cases.

Dynamic Semantics

18/3 {*AI95-00251-01*} {*AI05-0070-1*} The elaboration of an interface_type_definition creates the interface type and its first subtype.

18.a/3 **Discussion:** There is no other effect. An interface_list is made up of subtype_marks, which do not need to be elaborated, so the interface_list does not either. This is consistent with the handling of discriminant_parts.

19/5 NOTE {*AI95-00411-01*} {*AI12-0440-1*} Nonlimited interface types have predefined nonabstract equality operators. These can be overridden with user-defined abstract equality operators. Such operators will then require an explicit overriding for any nonabstract descendant of the interface.

Examples

20/2 {*AI95-00433-01*} *Example of a limited interface and a synchronized interface extending it:*

21/2
```
type Queue is limited interface;
procedure Append(Q : in out Queue; Person : in Person_Name) is abstract;
procedure Remove_First(Q      : in out Queue;
                       Person :    out Person_Name) is abstract;
function Cur_Count(Q : in Queue) return Natural is abstract;
function Max_Count(Q : in Queue) return Natural is abstract;
-- See 3.10.1 for Person_Name.
```

22/3
```
{AI05-0004-1} Queue_Error : exception;
-- Append raises Queue_Error if Cur_Count(Q) = Max_Count(Q)
-- Remove_First raises Queue_Error if Cur_Count(Q) = 0
```

23/2
```
type Synchronized_Queue is
                       synchronized interface and Queue; -- see 9.11
procedure Append_Wait(Q      : in out Synchronized_Queue;
                      Person : in     Person_Name) is abstract;
procedure Remove_First_Wait(Q      : in out Synchronized_Queue;
                            Person :    out Person_Name) is abstract;
```

24/2
```
...
```

```
procedure Transfer(From   : in out Queue'Class;                    25/2
                   To     : in out Queue'Class;
                   Number : in     Natural := 1) is
   Person : Person_Name;
begin
   for I in 1..Number loop
      Remove_First(From, Person);
      Append(To, Person);
   end loop;
end Transfer;
```

{*AI12-0442-1*} This defines a Queue interface defining a queue of people. (A similar design is possible to define any kind of queue simply by replacing Person_Name by an appropriate type.) The Queue interface has four dispatching operations, Append, Remove_First, Cur_Count, and Max_Count. The body of a class-wide operation, Transfer is also shown. Every nonabstract extension of Queue will provide implementations for at least its four dispatching operations, as they are abstract. Any object of a type derived from Queue can be passed to Transfer as either the From or the To operand. The two operands can be of different types in a given call. 26/5

{*AI12-0440-1*} The Synchronized_Queue interface inherits the four dispatching operations from Queue and adds two additional dispatching operations, which wait if necessary rather than raising the Queue_Error exception. This synchronized interface can only be implemented by a task or protected type, and as such ensures safe concurrent access. 27/5

{*AI95-00433-01*} *Example use of the interface:* 28/2

```
{AI05-0004-1} type Fast_Food_Queue is new Queue with record ...;    29/3
procedure Append(Q : in out Fast_Food_Queue; Person : in Person_Name);
procedure Remove_First(Q : in out Fast_Food_Queue;
                       Person : out Person_Name);
function Cur_Count(Q : in Fast_Food_Queue) return Natural;
function Max_Count(Q : in Fast_Food_Queue) return Natural;

...                                                                 30/2

Cashier, Counter : Fast_Food_Queue;                                 31/2

{AI12-0312-1} ...                                                   32/5
-- Add Casey (see 3.10.1) to the cashier's queue:
Append (Cashier, Casey);
-- After payment, move Casey to the sandwich counter queue:
Transfer (Cashier, Counter);
...
```

{*AI12-0442-1*} An interface such as Queue can be used directly as the parent of a new type (as shown here), or can be used as a progenitor when a type is derived. In either case, the primitive operations of the interface are inherited. For Queue, the implementation of the four inherited routines will necessarily be provided. Inside the call of Transfer, calls will dispatch to the implementations of Append and Remove_First for type Fast_Food_Queue. 33/5

{*AI95-00433-01*} *Example of a task interface:* 34/2

```
type Serial_Device is task interface;  -- see 9.1                   35/2
procedure Read (Dev : in Serial_Device; C : out Character) is abstract;
procedure Write(Dev : in Serial_Device; C : in  Character) is abstract;
```

The Serial_Device interface has two dispatching operations which are intended to be implemented by task entries (see 9.1). 36/2

Extensions to Ada 95

{*AI95-00251-01*} {*AI95-00345-01*} Interface types are new. They provide multiple inheritance of interfaces, similar to the facility provided in Java and other recent language designs. 36.a/2

Wording Changes from Ada 2005

36.b/3 {*AI05-0070-1*} **Correction:** Corrected the definition of elaboration for an interface_type_definition to match that of other type definitions.

3.10 Access Types

1 A value of an access type (an *access value*) provides indirect access to the object or subprogram it *designates*. Depending on its type, an access value can designate either subprograms, objects created by allocators (see 4.8), or more generally *aliased* objects of an appropriate type.

1.a **Discussion:** A name *denotes* an entity; an access value *designates* an entity. The "dereference" of an access value X, written "X.**all**", is a name that denotes the entity designated by X.

Language Design Principles

1.b/5 {*AI05-0299-1*} {*AI12-0449-1*} Access values should always be well defined (barring uses of certain unchecked features of Clause 13). In particular, uninitialized access variables should be prevented by compile-time rules.

Syntax

2/2 {*AI95-00231-01*} access_type_definition ::=
[null_exclusion] access_to_object_definition
| [null_exclusion] access_to_subprogram_definition

3 access_to_object_definition ::=
access [general_access_modifier] subtype_indication

4 general_access_modifier ::= **all** | **constant**

5 access_to_subprogram_definition ::=
access [**protected**] **procedure** parameter_profile
| **access** [**protected**] **function** parameter_and_result_profile

5.1/2 {*AI95-00231-01*} null_exclusion ::= **not null**

6/2 {*AI95-00231-01*} {*AI95-00254-01*} {*AI95-00404-01*} access_definition ::=
[null_exclusion] **access** [**constant**] subtype_mark
| [null_exclusion] **access** [**protected**] **procedure** parameter_profile
| [null_exclusion] **access** [**protected**] **function** parameter_and_result_profile

Static Semantics

7/1 {*8652/0012*} {*AI95-00062-01*} There are two kinds of access types, *access-to-object* types, whose values designate objects, and *access-to-subprogram* types, whose values designate subprograms. Associated with an access-to-object type is a *storage pool*; several access types may share the same storage pool. All descendants of an access type share the same storage pool. A storage pool is an area of storage used to hold dynamically allocated objects (called *pool elements*) created by allocators[; storage pools are described further in 13.11, "Storage Management"].

8 Access-to-object types are further subdivided into *pool-specific* access types, whose values can designate only the elements of their associated storage pool, and *general* access types, whose values can designate the elements of any storage pool, as well as aliased objects created by declarations rather than allocators, and aliased subcomponents of other objects.

8.a **Implementation Note:** The value of an access type will typically be a machine address. However, a value of a pool-specific access type can be represented as an offset (or index) relative to its storage pool, since it can point only to the elements of that pool.

{*AI95-00225-01*} {*AI95-00363-01*} {*AI05-0053-1*} {*AI05-0142-4*} {*AI05-0277-1*} {*AI12-0228-1*} {*AI12-0324-1*} A view of an object is defined to be *aliased* if it is defined by an object_declaration, component_definition, parameter_specification, or extended_return_object_declaration with the reserved word **aliased**, or by a renaming of an aliased view. In addition, the dereference of an access-to-object value denotes an aliased view, as does a view conversion (see 4.6) of an aliased view. A qualified_expression denotes an aliased view when the operand denotes an aliased view. The current instance of an immutably limited type (see 7.5) is defined to be aliased. Finally, a formal parameter or generic formal object of a tagged type is defined to be aliased. [Aliased views are the ones that can be designated by an access value.] 9/5

Term entry: aliased view — view of an object that can be designated by an access value 9.a/5
Note: Objects allocated by allocators are aliased. Objects can also be explicitly declared as aliased with the reserved word aliased. The Access attribute can be used to create an access value designating an aliased object.

Ramification: The current instance of a nonlimited type is not aliased. 9.b

The object created by an allocator is aliased, but not its subcomponents, except of course for those that themselves have **aliased** in their component_definition. 9.c

The renaming of an aliased object is aliased. 9.d

Slices are never aliased. See 4.1.2 for more discussion. 9.e

Reason: {*AI95-00225-01*} The current instance of a limited type is defined to be aliased so that an access discriminant of a component can be initialized with T'Access inside the definition of T. Note that we don't want this to apply to a type that could become nonlimited later within its immediate scope, so we require the full definition to be limited. 9.f/2

A formal parameter of a tagged type is defined to be aliased so that a (tagged) parameter X may be passed to an access parameter P by using P => X'Access. Access parameters are most important for tagged types because of dispatching-on-access-parameters (see 3.9.2). By restricting this to formal parameters, we minimize problems associated with allowing components that are not declared aliased to be pointed-to from within the same record. 9.g

A view conversion of an aliased view is aliased so that the type of an access parameter can be changed without first converting to a named access type. For example: 9.h

```
type T1 is tagged ...;
procedure P(X : access T1);
```
9.i

```
type T2 is new T1 with ...;
procedure P(X : access T2) is
begin
    P(T1(X.all)'Access);   -- hand off to T1's P
    . . .      -- now do extra T2-specific processing
end P;
```
9.j

This paragraph was deleted.{*AI95-00363-01*} 9.k/2

We considered making more kinds of objects aliased by default. In particular, any object of a by-reference type will pretty much have to be allocated at an addressable location, so it can be passed by reference without using bit-field pointers. Therefore, one might wish to allow the Access and Unchecked_Access attributes for such objects. However, private parts are transparent to the definition of "by-reference type", so if we made all objects of a by-reference type aliased, we would be violating the privacy of private parts. Instead, we would have to define a concept of "visibly by-reference" and base the rule on that. This seemed to complicate the rules more than it was worth, especially since there is no way to declare an untagged limited private type to be by-reference, since the full type might by nonlimited. 9.l/2

Discussion: Note that we do not use the term "aliased" to refer to formal parameters that are referenced through multiple access paths (see 6.2). 9.m

An access_to_object_definition defines an access-to-object type and its first subtype; the subtype_indication defines the *designated subtype* of the access type. If a general_access_modifier appears, then the access type is a general access type. If the modifier is the reserved word **constant**, then the type is an *access-to-constant type*[; a designated object cannot be updated through a value of such a type]. If the modifier is the reserved word **all**, then the type is an *access-to-variable type*[; a designated object can be both read and updated through a value of such a type]. If no general_access_modifier appears in the access_to_object_definition, the access type is a pool-specific access-to-variable type. 10

To be honest: The type of the designated subtype is called the *designated type*. 10.a

10.b **Reason:** The modifier **all** was picked to suggest that values of a general access type could point into "all" storage pools, as well as to objects declared aliased, and that "all" access (both read and update) to the designated object was provided. We couldn't think of any use for pool-specific access-to-constant types, so any access type defined with the modifier **constant** is considered a general access type, and can point into any storage pool or at other (appropriate) aliased objects.

10.c **Implementation Note:** The predefined generic Unchecked_Deallocation can be instantiated for any named access-to-variable type. There is no (language-defined) support for deallocating objects designated by a value of an access-to-constant type. Because of this, an allocator for an access-to-constant type can allocate out of a storage pool with no support for deallocation. Frequently, the allocation can be done at link-time, if the size and initial value are known then.

10.d **Discussion:** For the purpose of generic formal type matching, the relevant subclasses of access types are access-to-subprogram types, access-to-constant types, and (named) access-to-variable types, with its subclass (named) general access-to-variable types. Pool-specific access-to-variable types are not a separately matchable subclass of types, since they don't have any "extra" operations relative to all (named) access-to-variable types.

11 An access_to_subprogram_definition defines an access-to-subprogram type and its first subtype; the parameter_profile or parameter_and_result_profile defines the *designated profile* of the access type. There is a *calling convention* associated with the designated profile[; only subprograms with this calling convention can be designated by values of the access type.] By default, the calling convention is "*protected*" if the reserved word **protected** appears, and "Ada" otherwise. [See Annex B for how to override this default.]

11.a **Ramification:** The calling convention *protected* is in italics to emphasize that it cannot be specified explicitly by the user. This is a consequence of it being a reserved word.

11.b/2 **Implementation Note:** {*AI95-00254-01*} For a named access-to-subprogram type, the representation of an access value might include implementation-defined information needed to support up-level references — for example, a static link. The accessibility rules (see 3.10.2) ensure that in a "global-display-based" implementation model (as opposed to a static-link-based model), a named access-to-(unprotected)-subprogram value need consist only of the address of the subprogram. The global display is guaranteed to be properly set up any time the designated subprogram is called. Even in a static-link-based model, the only time a static link is definitely required is for an access-to-subprogram type declared in a scope nested at least two levels deep within subprogram or task bodies, since values of such a type might designate subprograms nested a smaller number of levels. For the normal case of a named access-to-subprogram type declared at the outermost (library) level, a code address by itself should be sufficient to represent the access value in many implementations.

11.c For access-to-protected-subprogram, the access values will necessarily include both an address (or other identification) of the code of the subprogram, as well as the address of the associated protected object. This could be thought of as a static link, but it will be needed even for global-display-based implementation models. It corresponds to the value of the "implicit parameter" that is passed into every call of a protected operation, to identify the current instance of the protected type on which they are to operate.

11.d Any Elaboration_Check is performed when a call is made through an access value, rather than when the access value is first "created" via a 'Access. For implementation models that normally put that check at the call-site, an access value will have to point to a separate entry point that does the check. Alternatively, the access value could point to a "subprogram descriptor" that consisted of two words (or perhaps more), the first being the address of the code, the second being the elaboration bit. Or perhaps more efficiently, just the address of the code, but using the trick that the descriptor is initialized to point to a Raise-Program-Error routine initially, and then set to point to the "real" code when the body is elaborated.

11.e For implementations that share code between generic instantiations, the extra level of indirection suggested above to support Elaboration_Checks could also be used to provide a pointer to the per-instance data area normally required when calling shared code. The trick would be to put a pointer to the per-instance data area into the subprogram descriptor, and then make sure that the address of the subprogram descriptor is loaded into a "known" register whenever an indirect call is performed. Once inside the shared code, the address of the per-instance data area can be retrieved out of the subprogram descriptor, by indexing off the "known" register.

11.f/2 *This paragraph was deleted.*{*AI95-00344-01*}

11.g/2 {*AI95-00254-01*} Note that access parameters of an anonymous access-to-subprogram type are permitted. Such parameters represent full "downward" closures, meaning that in an implementation that uses a per-task (global) display, the display will have to be passed as a hidden parameter, and reconstructed at the point of call.

12/3 {*AI95-00230-01*} {*AI95-00231-01*} {*AI95-00254-01*} {*AI05-0264-1*} An access_definition defines an anonymous general access type or an anonymous access-to-subprogram type. For a general access type,

the subtype_mark denotes its *designated subtype*; if the general_access_modifier **constant** appears, the type is an access-to-constant type; otherwise, it is an access-to-variable type. For an access-to-subprogram type, the parameter_profile or parameter_and_result_profile denotes its *designated profile*.

{*AI95-00230-01*} {*AI95-00231-01*} For each access type, there is a null access value designating no entity at all, which can be obtained by (implicitly) converting the literal **null** to the access type. [The null value of an access type is the default initial value of the type.] Nonnull values of an access-to-object type are obtained by evaluating an allocator[, which returns an access value designating a newly created object (see 3.10.2)], or in the case of a general access-to-object type, evaluating an attribute_reference for the Access or Unchecked_Access attribute of an aliased view of an object. Nonnull values of an access-to-subprogram type are obtained by evaluating an attribute_reference for the Access attribute of a nonintrinsic subprogram. 13/2

This paragraph was deleted.{*AI95-00231-01*} 13.a/2

This paragraph was deleted.{*AI95-00231-01*} 13.b/2

{*AI95-00231-01*} A null_exclusion in a construct specifies that the null value does not belong to the access subtype defined by the construct, that is, the access subtype *excludes null*. In addition, the anonymous access subtype defined by the access_definition for a controlling access parameter (see 3.9.2) excludes null. Finally, for a subtype_indication without a null_exclusion, the subtype denoted by the subtype_indication excludes null if and only if the subtype denoted by the subtype_mark in the subtype_indication excludes null. 13.1/2

Reason: {*AI95-00231-01*} An access_definition used in a controlling parameter excludes null because it is necessary to read the tag to dispatch, and null has no tag. We would have preferred to require **not null** to be specified for such parameters, but that would have been too incompatible with Ada 95 code to require. 13.c/2

{*AI95-00416-01*} Note that we considered imposing a similar implicit null exclusion for controlling access results, but chose not to do that, because there is no Ada 95 compatibility issue, and there is no automatic null check inherent in the use of a controlling access result. If a null check is necessary, it is because there is a dereference of the result, or because the value is passed to a parameter whose subtype excludes null. If there is no dereference of the result, a null return value is perfectly acceptable, and can be a useful indication of a particular status of the call. 13.d/2

{*8652/0013*} {*AI95-00012-01*} {*AI05-0264-1*} [All subtypes of an access-to-subprogram type are constrained.] The first subtype of a type defined by an access_definition or an access_to_object_definition is unconstrained if the designated subtype is an unconstrained array or discriminated subtype; otherwise, it is constrained. 14/3

Proof: The Legality Rules on range_constraints (see 3.5) do not permit the subtype_mark of the subtype_indication to denote an access-to-scalar type, only a scalar type. The Legality Rules on index_constraints (see 3.6.1) and discriminant_constraints (see 3.7.1) both permit access-to-composite types in a subtype_indication with such _constraints. Note that an access-to-access-to-composite is never permitted in a subtype_indication with a constraint. 14.a

Reason: {*AI95-00363-01*} Only composite_constraints are permitted for an access type, and only on access-to-composite types. A constraint on an access-to-scalar or access-to-access type might be violated due to assignments via other access paths that were not so constrained. By contrast, if the designated subtype is an array or discriminated type without defaults, the constraint could not be violated by unconstrained assignments, since array objects are always constrained, and discriminated objects are also constrained when the type does not have defaults for its discriminants. Constraints are not allowed on general access-to-unconstrained discriminated types if the type has defaults for its discriminants; constraints on pool-specific access types are usually allowed because allocated objects are usually constrained by their initial value. 14.b/2

Legality Rules

{*AI95-00231-01*} If a subtype_indication, discriminant_specification, parameter_specification, parameter_and_result_profile, object_renaming_declaration, or formal_object_declaration has a null_exclusion, the subtype_mark in that construct shall denote an access subtype that does not exclude null. 14.1/2

To be honest: {*AI95-00231-01*} This means “directly allowed in”; we are not talking about a null_exclusion that occurs in an access_definition in one of these constructs (for an access_definition, the subtype_mark in such an access_definition is not restricted). 14.c/2

14.d/2 **Reason:** {*AI95-00231-01*} This is similar to doubly constraining a composite subtype, which we also don't allow.

Dynamic Semantics

15/2 {*AI95-00231-01*} A composite_constraint is *compatible* with an unconstrained access subtype if it is compatible with the designated subtype. A null_exclusion is compatible with any access subtype that does not exclude null. An access value *satisfies* a composite_constraint of an access subtype if it equals the null value of its type or if it designates an object whose value satisfies the constraint. An access value satisfies an exclusion of the null value if it does not equal the null value of its type.

16 The elaboration of an access_type_definition creates the access type and its first subtype. For an access-to-object type, this elaboration includes the elaboration of the subtype_indication, which creates the designated subtype.

17/2 {*AI95-00230-01*} {*AI95-00254-01*} The elaboration of an access_definition creates an anonymous access type.

18 NOTE 1 Access values are called “pointers” or “references” in some other languages.

19 NOTE 2 Each access-to-object type has an associated storage pool; several access types can share the same pool. An object can be created in the storage pool of an access type by an allocator (see 4.8) for the access type. A storage pool (roughly) corresponds to what some other languages call a “heap”. See 13.11 for a discussion of pools.

20 NOTE 3 Only index_constraints and discriminant_constraints can be applied to access types (see 3.6.1 and 3.7.1).

Examples

21 *Examples of access-to-object types:*

22/4
```
{AI95-00433-01} {AI12-0056-1} type Frame is access Matrix;  -- see 3.6
type Peripheral_Ref is not null access Peripheral;  -- see 3.8.1
type Binop_Ptr is access all Binary_Operation'Class;
                                -- general access-to-class-wide, see 3.9.1
```

23 *Example of an access subtype:*

24
```
subtype Drum_Ref is Peripheral_Ref(Drum); -- see 3.8.1
```

25 *Example of an access-to-subprogram type:*

26
```
type Message_Procedure is access procedure (M : in String := "Error!");
procedure Default_Message_Procedure(M : in String);
Give_Message : Message_Procedure := Default_Message_Procedure'Access;
...
procedure Other_Procedure(M : in String);
...
Give_Message := Other_Procedure'Access;
...
Give_Message("File not found.");
                          -- call with parameter (.all is optional)
Give_Message.all;         -- call with no parameters
```

Extensions to Ada 83

26.a The syntax for access_type_definition is changed to support general access types (including access-to-constants) and access-to-subprograms. The syntax rules for general_access_modifier and access_definition are new.

Wording Changes from Ada 83

26.b/3 {*AI05-0190-1*} We use the term "storage pool" to talk about the data area from which allocation takes place. The term "collection" is only used for finalization. ("Collection" and "storage pool" are not the same thing because multiple unrelated access types can share the same storage pool; see 13.11 for more discussion.)

Inconsistencies With Ada 95

26.c/2 {*AI95-00231-01*} Access discriminants and noncontrolling access parameters no longer exclude null. A program which passed **null** to such an access discriminant or access parameter and expected it to raise Constraint_Error may fail when

compiled with Ada 2005. One hopes that there no such programs outside of the ACATS. (Of course, a program which actually wants to pass **null** will work, which is far more likely.)

{*AI95-00363-01*} Most unconstrained aliased objects with defaulted discriminants are no longer constrained by their initial values. This means that a program that raised Constraint_Error from an attempt to change the discriminants will no longer do so. The change only affects programs that depended on the raising of Constraint_Error in this case, so the inconsistency is unlikely to occur outside of the ACATS. This change may however cause compilers to implement these objects differently, possibly taking additional memory or time. This is unlikely to be worse than the differences caused by any major compiler upgrade. 26.d/2

Incompatibilities With Ada 95

{*AI95-00225-01*} **Amendment Correction:** The rule defining when a current instance of a limited type is considered to be aliased has been tightened to apply only to types that cannot become nonlimited. A program that attempts to take 'Access of the current instance of a limited type that can become nonlimited will be illegal in Ada 2005. While original Ada 95 allowed the current instance of any limited type to be treated as aliased, this was inconsistently implemented in compilers, and was likely to not work as expected for types that are ultimately nonlimited. 26.e/2

{*AI12-0289-1*} **Correction:** Static matching (see 4.9.1) requires that both anonymous access types exclude null; and full conformance requires statically matching subtypes. Because of the definition of “excludes null” given in this subclause, an access parameter that designates an untagged private type P (which does not exclude null) does not match its completion if P is completed with a tagged type (in that case, the parameter is controlling and thus excludes null, regardless of whether there is an explicit null exclusion on the body). Similar considerations apply in contexts where mode conformance and subtype conformance are required (for instance, subprogram renaming). 26.f/5

Extensions to Ada 95

{*AI95-00231-01*} The null_exclusion is new. It can be used in both anonymous and named access type definitions. It is most useful to declare that parameters cannot be **null**, thus eliminating the need for checks on use. 26.g/2

{*AI95-00231-01*} {*AI95-00254-01*} {*AI95-00404-01*} The kinds of anonymous access types allowed were increased by adding anonymous access-to-constant and anonymous access-to-subprogram types. Anonymous access-to-subprogram types used as parameters allow passing of subprograms at any level. 26.h/2

Wording Changes from Ada 95

{*8652/0012*} {*AI95-00062-01*} **Corrigendum:** Added accidentally-omitted wording that says that a derived access type shares its storage pool with its parent type. This was clearly intended, both because of a note in 3.4, and because anything else would have been incompatible with Ada 83. 26.i/2

{*8652/0013*} {*AI95-00012-01*} **Corrigendum:** Fixed typographical errors in the description of when access types are constrained. 26.j/2

{*AI95-00230-01*} The wording was fixed to allow allocators and the literal **null** for anonymous access types. The former was clearly intended by Ada 95; see the Implementation Advice in 13.11. 26.k/2

{*AI95-00363-01*} The rules about aliased objects being constrained by their initial values now apply only to allocated objects, and thus have been moved to 4.8, “Allocators”. 26.l/2

Wording Changes from Ada 2005

{*AI05-0053-1*} {*AI05-0277-1*} **Correction:** The rule about a current instance being aliased now is worded in terms of immutably limited types. Wording was also added to make extended return object declarations that have the keyword **aliased** be considered aliased. This latter was a significant oversight in Ada 2005 — technically, the keyword **aliased** had no effect. But of course implementations followed the intent, not the letter of the Reference Manual. 26.m/3

{*AI05-0142-4*} Explicitly aliased parameters (see 6.1) are defined to be aliased. 26.n/3

Extensions to Ada 2012

{*AI12-0228-1*} **Correction:** A qualified_expression of an aliased object is now also aliased. 26.o/5

3.10.1 Incomplete Type Declarations

There are no particular limitations on the designated type of an access type. In particular, the type of a component of the designated type can be another access type, or even the same access type. This permits mutually dependent and recursive access types. An incomplete_type_declaration can be used to 1

introduce a type to be used as a designated type, while deferring its full definition to a subsequent full_type_declaration.

Syntax

2/2 {*AI95-00326-01*} incomplete_type_declaration ::=
type defining_identifier [discriminant_part] [**is tagged**];

Static Semantics

2.1/4 {*AI95-00326-01*} {*AI12-0137-1*} An incomplete_type_declaration declares an *incomplete view* of a type and its first subtype; the first subtype is unconstrained if a discriminant_part appears. If the incomplete_type_declaration includes the reserved word **tagged**, it declares a *tagged incomplete view*. If *T* denotes a tagged incomplete view, then *T*'Class denotes a tagged incomplete view. [An incomplete view of a type is a limited view of the type (see 7.5).]

2.2/2 {*AI95-00326-01*} Given an access type *A* whose designated type *T* is an incomplete view, a dereference of a value of type *A* also has this incomplete view except when:

2.a/3 **Discussion:** {*AI05-0208-1*} Whether the designated type is an incomplete view (and thus whether this set of rules applies) is determined by the view of the type at the declaration of the access type; it does not change during the life of the type.

2.3/2 - it occurs within the immediate scope of the completion of *T*, or

2.4/3 - {*AI05-0208-1*} it occurs within the scope of a nonlimited_with_clause that mentions a library package in whose visible part the completion of *T* is declared, or

2.5/3 - {*AI05-0208-1*} it occurs within the scope of the completion of *T* and *T* is an incomplete view declared by an incomplete_type_declaration.

2.6/3 {*AI05-0162-1*} In these cases, the dereference has the view of *T* visible at the point of the dereference.

2.b/2 **Discussion:** We need the "in whose visible part" rule so that the second rule doesn't trigger in the body of a package with a **with** of a child unit:

2.c/2
```
package P is
private
   type T;
   type PtrT is access T;
end P;
```

2.d/2
```
private package P.C is
   Ptr : PtrT;
end P.C;
```

2.e/3
```
{AI05-0005-1} with P.C;
package body P is
     -- Ptr.all'Size is not legal here, but we are within the scope
     -- of a nonlimited_with_clause for P.
type T is ...
     --   Ptr.all'Size is legal here.
end P;
```

2.7/3 {*AI95-00412-01*} {*AI05-0162-1*} {*AI05-0208-1*} Similarly, if a subtype_mark denotes a subtype_declaration defining a subtype of an incomplete view *T*, the subtype_mark denotes an incomplete view except under the same three circumstances given above, in which case it denotes the view of *T* visible at the point of the subtype_mark.

Legality Rules

3/3 {*AI05-0162-1*} An incomplete_type_declaration requires a completion, which shall be a type_declaration other than an incomplete_type_declaration. [If the incomplete_type_declaration occurs immediately within either the visible part of a package_specification or a declarative_part, then the type_declaration shall occur later and immediately within this visible part or declarative_part. If the incomplete_type_declaration occurs immediately within the private part of a given package_-

specification, then the type_declaration shall occur later and immediately within either the private part itself, or the declarative_part of the corresponding package_body.]

Proof: This is implied by the next AARM-only rule, plus the rules in 3.11.1, "Completions of Declarations" which require a completion to appear later and immediately within the same declarative region. 3.a

To be honest: If the incomplete_type_declaration occurs immediately within the visible part of a package_specification, then the completing type_declaration shall occur immediately within this visible part. 3.b

To be honest: If the implementation supports it, an incomplete_type_declaration can be imported (using aspect Import, see B.1), in which case no explicit completion is allowed. 3.c

{*AI95-00326-01*} {*AI05-0162-1*} If an incomplete_type_declaration includes the reserved word **tagged**, then a type_declaration that completes it shall declare a tagged type. If an incomplete_type_declaration has a known_discriminant_part, then a type_declaration that completes it shall have a fully conforming (explicit) known_discriminant_part (see 6.3.1). [If an incomplete_type_declaration has no discriminant_part (or an unknown_discriminant_part), then a corresponding type_declaration is nevertheless allowed to have discriminants, either explicitly, or inherited via derivation.] 4/3

{*AI95-00326-01*} A name that denotes an incomplete view of a type may be used as follows: 5/2

- {*AI05-0098-1*} as the subtype_mark in the subtype_indication of an access_to_object_definition; [the only form of constraint allowed in this subtype_indication is a discriminant_constraint [(a null_exclusion is not allowed)];] 6/3

  Implementation Note: We now allow discriminant_constraints even if the full type is deferred to the package body. However, there is no particular implementation burden because we have dropped the concept of the dependent compatibility check. In other words, we have effectively repealed AI83-00007. 6.a

- {*AI95-00326-01*} {*AI95-00412-01*} as the subtype_mark in the subtype_indication of a subtype_declaration; the subtype_indication shall not have a null_exclusion or a constraint; 7/2

- {*AI95-00326-01*} {*AI05-0151-1*} as the subtype_mark in an access_definition for an access-to-object type; 8/3

  To be honest: This does not mean any random subtype_mark in a construct that makes up an access_definition, such as a formal_part, just the one given directly in the syntax of access_definition. 8.a/2

- {*AI05-0151-1*} as the subtype_mark defining the subtype of a parameter or result in a profile occurring within a basic_declaration; 8.1/3

  Ramification: But not in the profile for a body or entry. 8.b/3

- {*AI05-0213-1*} as a generic actual parameter whose corresponding generic formal parameter is a formal incomplete type (see 12.5.1). 8.2/3

{*AI95-00326-01*} If such a name denotes a tagged incomplete view, it may also be used: 8.3/2

- {*AI95-00326-01*} {*AI05-0151-1*} as the subtype_mark defining the subtype of a parameter in the profile for a subprogram_body, entry_body, or accept_statement; 8.4/3

- {*AI95-00326-01*} as the prefix of an attribute_reference whose attribute_designator is Class; such an attribute_reference is restricted to the uses allowed here; it denotes a tagged incomplete view. 9/2

  This paragraph was deleted.{*AI95-00326-01*} 9.a/2

This paragraph was deleted.{*AI95-00326-01*} {*AI05-0151-1*} 9.1/3

- *This paragraph was deleted.*{*AI95-00326-01*} {*AI05-0098-1*} {*AI05-0151-1*} 9.2/3

  This paragraph was deleted. 9.b/3

{*AI95-00326-01*} If any of the above uses occurs as part of the declaration of a primitive subprogram of the incomplete view, and the declaration occurs immediately within the private part of a package, then the completion of the incomplete view shall also occur immediately within the private part; it shall not be deferred to the package body. 9.3/2

9.c/2 **Reason:** This fixes a hole in Ada 95 where a dispatching operation with an access parameter could be declared in a private part and a dispatching call on it could occur in a child even though there is no visibility on the full type, requiring access to the controlling tag without access to the representation of the type.

9.4/2 {*AI95-00326-01*} No other uses of a name that denotes an incomplete view of a type are allowed.

10/3 {*AI95-00326-01*} {*AI05-0151-1*} A prefix that denotes an object shall not be of an incomplete view. An actual parameter in a call shall not be of an untagged incomplete view. The result object of a function call shall not be of an incomplete view. A prefix shall not denote a subprogram having a formal parameter of an untagged incomplete view, nor a return type that is an incomplete view.

10.a/2 **Reason:** We used to disallow all dereferences of an incomplete type. Now we only disallow such dereferences when used as a prefix. Dereferences used in other contexts do not pose a problem since normal type matching will preclude their use except when the full type is "nearby" as context (for example, as the expected type).

10.b/2 This also disallows prefixes that are directly of an incomplete view. For instance, a parameter *P* can be declared of a tagged incomplete type, but we don't want to allow *P*'Size, *P*'Alignment, or the like, as representation values aren't known for an incomplete view.

10.c/2 We say "denotes an object" so that prefixes that directly name an incomplete view are not covered; the previous rules cover such cases, and we certainly don't want to ban Incomp'Class.

10.d/3 {*AI05-0151-1*} As subprogram profiles now may include any kind of incomplete type, we also disallow passing objects of untagged incomplete types in subprogram calls (as the parameter passing method is not known as it is for tagged types) and disallow returning any sort of incomplete objects (since we don't know how big they are).

10.1/5 {*AI12-0155-1*} The controlling operand or controlling result of a dispatching call shall not be of an incomplete view if the operand or result is dynamically tagged.

10.e/5 **Reason:** This rule is needed to prevent the following case:

10.f/5
```
package Pack is
   type T is tagged;
   function F return access T'Class;
   function G (X : access T) return Integer;
   I : Integer := G (F);                  -- Illegal by above rule.
   type T is tagged null record;
end Pack;
```

10.g/5 If this was not illegal, the compiler would have to generate a dispatching call on G without necessarily knowing where the tag of type T is stored (the completion of T might not be until the body of Pack). The fact that any such call will raise Program_Error does not absolve us of detecting the problem; see the Language Design Principles in 13.14.

Paragraph 11 was deleted.

Dynamic Semantics

12 The elaboration of an incomplete_type_declaration has no effect.

12.a **Reason:** An incomplete type has no real existence, so it doesn't need to be "created" in the usual sense we do for other types. It is roughly equivalent to a "forward;" declaration in Pascal. Private types are different, because they have a different set of characteristics from their full type.

13/5 NOTE 1 {*AI12-0447-1*} Within a declarative_part, an incomplete_type_declaration and a corresponding full_type_declaration cannot be separated by an intervening body. This is because, by the rules given in 13.14, a type is illegal if it is not completely defined before it is frozen, and a body freezes all types declared prior to it in the same declarative_part.

13.1/3 NOTE 2 {*AI05-0151-1*} {*AI05-0269-1*} A name that denotes an object of an incomplete view is defined to be of a limited type. Hence, the target of an assignment statement cannot be of an incomplete view.

Examples

14 *Example of a recursive type:*

15
```
type Cell;  -- incomplete type declaration
type Link is access Cell;
```

```
type Cell is                                                                    16
   record
      Value  : Integer;
      Succ   : Link;
      Pred   : Link;
   end record;

Head   : Link  := new Cell'(0, null, null);                                     17
Next   : Link  := Head.Succ;
```

Examples of mutually dependent access types: 18

```
{AI95-00433-01} type Person(<>);     -- incomplete type declaration             19/2
type Car is tagged; -- incomplete type declaration

{AI95-00433-01} type Person_Name is access Person;                              20/2
type Car_Name    is access all Car'Class;

{AI95-00433-01} type Car is tagged                                              21/2
   record
      Number  : Integer;
      Owner   : Person_Name;
   end record;

type Person(Sex : Gender) is                                                    22
   record
      Name     : String(1 .. 20);
      Birth    : Date;
      Age      : Integer range 0 .. 130;
      Vehicle  : Car_Name;
      case Sex is
         when M => Wife           : Person_Name(Sex => F);
         when F => Husband        : Person_Name(Sex => M);
      end case;
   end record;

{AI12-0312-1} My_Car, Your_Car, Next_Car : Car_Name := new Car;  -- see 4.8    23/5
Casey : Person_Name := new Person(M);
   ...
Casey.Vehicle := Your_Car;
```

Extensions to Ada 83

The full_type_declaration that completes an incomplete_type_declaration may have a known_discriminant_part even if the incomplete_type_declaration does not. 23.a

A discriminant_constraint may be applied to an incomplete type, even if its completion is deferred to the package body, because there is no “dependent compatibility check” required any more. Of course, the constraint can be specified only if a known_discriminant_part was given in the incomplete_type_declaration. As mentioned in the previous paragraph, that is no longer required even when the full type has discriminants. 23.b/1

Wording Changes from Ada 83

Dereferences producing incomplete types were not explicitly disallowed in RM83, though AI83-00039 indicated that it was not strictly necessary since troublesome cases would result in Constraint_Error at run time, since the access value would necessarily be null. However, this introduces an undesirable implementation burden, as illustrated by Example 4 of AI83-00039: 23.c

```
package Pack is                                                                 23.d
    type Pri is private;
private
    type Sep;
    type Pri is access Sep;
    X : Pri;
end Pack;

package body Pack is -- Could be separately compiled!                           23.e
    type Sep is ...;
    X := new Sep;
end Pack;
```

23.f
```
pragma Elaborate(Pack);
private package Pack.Child is
    I : Integer := X.all'Size; -- Legal, by AI-00039.
end Pack.Child;
```

23.g Generating code for the above example could be a serious implementation burden, since it would require all aliased objects to store size dope, and for that dope to be in the same format for all kinds of types (or some other equivalently inefficient implementation). On the contrary, most implementations allocate dope differently (or not at all) for different designated subtypes.

Incompatibilities With Ada 95

23.h/2 {*AI95-00326-01*} It is now illegal to use an incomplete view (type) as the parameter or result of an access-to-subprogram type unless the incomplete view is completed in the same declaration list as the use. This was allowed in Ada 95 for incomplete types where the completion was deferred to the body. By disallowing this rare use of incomplete views, we can allow the use of incomplete views in many more places, which is especially valuable for limited views.

23.i/2 {*AI95-00326-01*} It is now illegal to use an incomplete view (type) in a primitive subprogram of the type unless the incomplete view is completed in the package specification. This was allowed in Ada 95 for incomplete types where the completion was deferred to the body (the use would have to be in an access parameter). This incompatibility was caused by the fix for the hole noted in Legality Rules above.

Extensions to Ada 95

23.j/2 {*AI95-00326-01*} Tagged incomplete types are new. They are allowed in parameter declarations as well as the usual places, as tagged types are always by-reference types (and thus there can be no code generation issue).

23.k/2 {*AI95-00412-01*} A subtype_declaration can be used to give a new name to an incomplete view of a type. This is valuable to give shorter names to entities imported with a limited_with_clause.

Wording Changes from Ada 95

23.l/2 {*AI95-00326-01*} The description of incomplete types as *incomplete views* is new. Ada 95 defined these as separate types, but neglected to give any rules for matching them with other types. Luckily, implementers did the right thing anyway. This change also makes it easier to describe the meaning of a limited view.

Extensions to Ada 2005

23.m/3 {*AI05-0098-1*} **Correction:** Fixed the definition so that an anonymous access-to-subprogram type can use an incomplete view in the same way that a named access-to-subprogram type can.

23.n/3 {*AI05-0151-1*} Incomplete types now can be used in subprogram declarations. The type has to be complete before any calls or the body is declared. This reduces the places where access types are required for types imported from limited views of packages.

23.o/3 {*AI05-0162-1*} Incomplete types now can be completed by private types and private extensions. Since this can already happen for limited views, there is no remaining reason to disallow it for explicitly declared incomplete types.

Wording Changes from Ada 2005

23.p/3 {*AI05-0208-1*} **Correction:** Changed the rules of uses of dereferences of incomplete views such that it does not introduce an unintentional incompatibility with Ada 83 and Ada 95.

23.q/3 {*AI05-0213-1*} Incomplete types now can be used as actuals to formal incomplete types (see 12.5.1).

Wording Changes from Ada 2012

23.r/5 {*AI12-0137-1*} {*AI12-0005-1*} **Corrigendum:** Changed the wording to clarify that the class-wide type associated with a specific tagged type that has an incomplete view is also an incomplete view. While this was previously undefined, an interpretation where it is not an incomplete view leads to semantic nonsense, and thus we don't consider this a potential incompatibility, as compilers most likely are doing the right thing.

23.s/5 {*AI12-0155-1*} **Correction:** Added a rule preventing a dispatching call on an incomplete type before the completion. This is not an incompatibility as previously this was prevented by the freezing rules. However, that violated the intended design of the freezing rules, so we've now changed to an explicit Legality Rule here, and eliminated the associated freezing rules.

3.10.2 Operations of Access Types

[The attribute Access is used to create access values designating aliased objects and nonintrinsic subprograms. The "accessibility" rules prevent dangling references (in the absence of uses of certain unchecked features — see Clause 13).] 1

Language Design Principles

It should be possible for an access value to designate an object declared by an object declaration, or a subcomponent thereof. In implementation terms, this means pointing at stack-allocated and statically allocated data structures. However, dangling references should be prevented, primarily via compile-time rules, so long as features like Unchecked_Access and Unchecked_Deallocation are not used. 1.a

In order to create such access values, we require that the access type be a general access type, that the designated object be aliased, and that the accessibility rules be obeyed. 1.b

Name Resolution Rules

{*AI95-00235-01*} For an attribute_reference with attribute_designator Access (or Unchecked_Access — see 13.10), the expected type shall be a single access type *A* such that: 2/2

- {*AI95-00235-01*} *A* is an access-to-object type with designated type *D* and the type of the prefix is *D*'Class or is covered by *D*, or 2.1/2
- {*AI95-00235-01*} *A* is an access-to-subprogram type whose designated profile is type conformant with that of the prefix. 2.2/2

{*AI95-00235-01*} [The prefix of such an attribute_reference is never interpreted as an implicit_dereference or a parameterless function_call (see 4.1.4).] The designated type or profile of the expected type of the attribute_reference is the expected type or profile for the prefix. 2.3/2

Discussion: Saying that the expected type shall be a "single access type" is our "new" way of saying that the type has to be determinable from context using only the fact that it is an access type. See 4.2 and 8.6. Specifying the expected profile only implies type conformance. The more stringent subtype conformance is required by a Legality Rule. This is the only Resolution Rule that applies to the name in a prefix of an attribute_reference. In all other cases, the name has to be resolved without using context. See 4.1.4. 2.a

{*AI95-00235-01*} Saying "single access type" is a bit of a fudge. Both the context and the prefix may provide both multiple types; "single" only means that a single, specific interpretation must remain after resolution. We say "single" here to trigger the Legality Rules of 8.6. The resolution of an access attribute is similar to that of an assignment_statement. For example: 2.b/2

```
type Int_Ptr is access all Integer;                                    2.c/2
type Char_Ptr is access all Character;
type Float_Ptr is access all Float;

function Zap (Val : Int_Ptr) return Float;   -- (1)                    2.d/2
function Zap (Val : Float_Ptr) return Float; -- (2)
function Zop return Int_Ptr;  -- (3)
function Zop return Char_Ptr; -- (4)

Result : Float := Zap (Zop.all'Access); -- Resolves to Zap (1) and Zop (3).   2.e/2
```

Static Semantics

{*AI95-00162-01*} {*AI12-0406-1*} [The accessibility rules, which prevent dangling references, are written in terms of *accessibility levels*, which reflect the run-time nesting of *masters*. As explained in 7.6.1, a master is the execution of a certain construct (called a *master construct*), such as a subprogram_body. An accessibility level is *deeper than* another if it is more deeply nested at run time. For example, an object declared local to a called subprogram has a deeper accessibility level than an object declared local to the calling subprogram. The accessibility rules for access types require that the accessibility level of an object designated by an access value be no deeper than that of the access type. This ensures that the object will live at least as long as the access type, which in turn ensures that the access value cannot later designate 3/5

an object that no longer exists. The Unchecked_Access attribute may be used to circumvent the accessibility rules.]

3.a/3 **Discussion:** {*AI05-0005-1*} The Unchecked_Access attribute acts as if the object was declared at library-level; this applies even when it is used as the value of anonymous access type. See 13.10.

3.b/3 Subclause 3.10.2, home of the accessibility rules, is informally known as the "Heart of Darkness" amongst the maintainers of Ada. Woe unto all who enter here (well, at least unto anyone that needs to understand any of these rules).

3.b.1/5 **Term entry: accessibility level** — representation of the lifetime of an entity in terms of the level of dynamic nesting within which the entity is known to exist

4 [A given accessibility level is said to be *statically deeper* than another if the given level is known at compile time (as defined below) to be deeper than the other for all possible executions. In most cases, accessibility is enforced at compile time by Legality Rules. Run-time accessibility checks are also used, since the Legality Rules do not cover certain cases involving access parameters and generic packages.]

5/5 {*AI12-0345-1*} Each master, and each entity and view created by it, has an accessibility level; when two levels are defined to be the same, the accessibility levels of the two associated entities are said to be *tied* to each other. Accessibility levels are defined as follows:

6 - The accessibility level of a given master is deeper than that of each dynamically enclosing master, and deeper than that of each master upon which the task executing the given master directly depends (see 9.3).

7/5 - {*AI95-00162-01*} {*AI95-00416-01*} {*AI05-0235-1*} {*AI12-0067-1*} {*AI12-0089-1*} {*AI12-0345-1*} An entity or view defined by a declaration and created as part of its elaboration has the same accessibility level as the innermost master of the declaration except in the cases of renaming and derived access types described below. A formal parameter of a callable entity has the same accessibility level as the master representing the invocation of the entity.

7.a/2 **Reason:** {*AI95-00416-01*} This rule defines the "normal" accessibility of entities. In the absence of special rules below, we intend for this rule to apply.

7.b/5 **Discussion:** {*AI95-00416-01*} {*AI12-0005-1*} {*AI12-0005-1*} This rule defines the accessibility of all named access types, as well as the accessibility level of anonymous access types defined by a component_definition Special rules exist for the accessibility level of other anonymous types. Components whose (anonymous) type is defined by an access_definition have accessibility levels corresponding to named access types defined at the same point.

7.c/2 **Ramification:** {*AI95-00230-01*} Because accessibility level is determined by where the access_definition is elaborated, for a type extension, the anonymous access types of components (other than access discriminants) inherited from the parent have the same accessibility as they did in the parent; those in the extension part have the accessibility determined by the scope where the type extension is declared. Similarly, the types of the nondiscriminant access components of a derived untagged type have the same accessibility as they did in the parent.

7.d/3 **To be honest:** {*AI05-0235-1*} We use "invocation of" in the parameter case as a master is formally an execution of something. But we mean this to be interpreted statically (for instance, as the body of the subprogram) for the purposes of computing "statically deeper than" (see below).

7.e/3 **Ramification:** {*AI05-0235-1*} Note that accessibility can differ depending on the view of an object (for both static and dynamic accessibility). For instance, the accessibility level of a formal parameter may be different than the accessibility level of the corresponding actual parameter. This occurs in other cases as well.

7.f/3 **Reason:** {*AI05-0235-1*} We define the (dynamic) accessibility of formal parameters in order that it does not depend on the parameter passing model (by-reference or by-copy) as that is implementation defined. Otherwise, there would be a portability issue.

8/5 - {*AI12-0371-1*} The accessibility level of a view of an object or subprogram defined by a renaming_declaration is the same as that of the renamed view, unless the renaming is of a formal subprogram, in which case the accessibility level is that of the instance.

9/2 - {*AI95-00416-01*} The accessibility level of a view conversion, qualified_expression, or parenthesized expression, is the same as that of the operand.

9.1/5 - {*AI05-0188-1*} {*AI12-0292-1*} The accessibility level of a conditional_expression (see 4.5.7) is the accessibility level of the evaluated *dependent*_expression.

- {*AI12-0236-1*} The accessibility level of a declare_expression (see 4.5.9) is the accessibility level of the *body*_expression. 9.2/5
- {*AI95-00318-02*} {*AI95-00416-01*} {*AI05-0234-1*} {*AI12-0027-1*} The accessibility level of an aggregate that is used (in its entirety) to directly initialize part of an object is that of the object being initialized. In other contexts, the accessibility level of an aggregate is that of the innermost master that evaluates the aggregate. Corresponding rules apply to a value conversion (see 4.6). 10/4
- {*AI05-0234-1*} The accessibility level of the result of a function call is that of the *master of the function call*, which is determined by the point of call as follows: 10.1/3
  - {*AI05-0234-1*} {*AI12-0402-1*} If the result type at the point of the function (or access-to-function type) declaration is a composite type, and the result is used (in its entirety) to directly initialize part of an object, the master is that of the object being initialized. In the case where the initialized object is a coextension (see below) that becomes a coextension of another object, the master is that of the eventual object to which the coextension will be transferred. 10.2/5

    To be honest: {*AI95-00416-01*} The first sentence is talking about a static use of the entire return object — a slice that happens to be the entire return object doesn't count. On the other hand, this is intended to allow parentheses and qualified_expressions. 10.a/2

    Ramification: {*AI95-00416-01*} {*AI05-0234-1*} If the function is used as a prefix, this bullet does not apply. Similarly, an assignment_statement is not an initialization of an object, so this bullet does not apply. 10.b/3

    Reason: {*AI12-0402-1*} We restrict the above rule to apply only if the function result is declared to be of a composite type. This makes sense since only results of a composite type (or of an anonymous access type, which are handled separately below) are potentially affected by the master of the function call. Note that a private type is considered composite, so this result assumes the worst for a function returning a private type. We could have made the rule more complex, depending on whether the result might be built in place, or might have an access discriminant, but we chose to keep the rule simpler. The wording says “the result type at the point of function declaration is a composite type”, to make it clear that this depends on the properties at the point of the declaration of the function, rather than properties that might be known at the point of call or inside the function body. 10.b.1/5
  - {*AI05-0234-1*} {*AI12-0278-1*} {*AI12-0390-1*} If the result is of an anonymous access type and is converted to a (named or anonymous) access type, the master is determined following the rules given below for determining the master of an object created by an allocator (even if the access result is of an access-to-subprogram type); 10.3/5

    Ramification: {*AI12-0278-1*} The conversion can be an explicit type conversion, or an implicit subtype conversion (these happen when anonymous access types are allowed to match named general access types, see 8.6). 10.b.2/5
  - *This paragraph was deleted.*{*AI05-0234-1*} {*AI12-0390-1*} 10.4/5
  - {*AI05-0234-1*} {*AI12-0345-1*} {*AI12-0372-1*} If the call itself defines the result of a function *F*, or has an accessibility level that is tied to the result of such a function *F*, then the master of the call is that of the master of the call invoking *F*; 10.5/5
  - {*AI05-0234-1*} In other cases, the master of the call is that of the innermost master that evaluates the function call. 10.6/3

    Ramification: {*AI95-00318-02*} {*AI95-00416-01*} The “innermost master which evaluated the function call” does not include the function call itself (which might be a master). 10.c/2

    {*AI95-00318-02*} {*AI95-00416-01*} We really mean the innermost master here, which could be a very short lifetime. Consider a function call used as a parameter of a procedure call. In this case the innermost master which evaluated the function call is the procedure call. 10.d/2

    Ramification: {*AI05-0234-1*} These rules do not mention whether the result object is built-in-place (see 7.6). In particular, in the case where building in place is optional, the choice whether or not to build-in-place has no effect on masters, lifetimes, or accessibility. 10.d.1/3

    Implementation Note: {*AI05-0234-1*} There are several cases where the implementation may have to pass in the accessibility level of the result object on a call, to support later rules where the accessibility level comes from the master of the call: 10.d.2/3
    - when the function result may have a part with access discriminants; 10.d.3/3

10.d.4/3 • when the function result type is an anonymous access type;

10.d.5/5 • {*AI12-0345-1*} when the function result is built-in-place.

10.d.6/5 • *This paragraph was deleted.*{*AI12-0345-1*}

10.d.7/3 In particular, this implies passing a level parameter when the result type is class-wide, since descendants may add access discriminants. For most implementations this will mean that functions with controlling results will also need a level parameter.

10.7/3 {*AI05-0284-1*} In the case of a call to a function whose result type is an anonymous access type, the accessibility level of the type of the result of the function call is also determined by the point of call as described above.

10.8/3 • {*AI95-00416-01*} Within a return statement, the accessibility level of the return object is that of the execution of the return statement. If the return statement completes normally by returning from the function, then prior to leaving the function, the accessibility level of the return object changes to be a level determined by the point of call, as does the level of any coextensions (see below) of the return object.

10.e/2 **Reason:** We define the accessibility level of the return object during the return statement to be that of the return statement itself so that the object may be designated by objects local to the return statement, but not by objects outside the return statement. In addition, the intent is that the return object gets finalized if the return statement ends without actually returning (for example, due to propagating an exception, or a goto). For a normal return, of course, no finalization is done before returning.

11 • The accessibility level of a derived access type is the same as that of its ultimate ancestor.

11.1/2 • {*AI95-00230-01*} The accessibility level of the anonymous access type defined by an access_definition of an object_renaming_declaration is the same as that of the renamed view.

11.2/5 • {*AI12-0156-1*} The accessibility level of the anonymous access type defined by an access_definition of a loop_parameter_subtype_indication is that of the loop parameter.

12/2 • {*AI95-00230-01*} {*AI95-00416-01*} The accessibility level of the anonymous access type of an access discriminant in the subtype_indication or qualified_expression of an allocator, or in the expression or return_subtype_indication of a return statement is determined as follows:

12.1/2 • If the value of the access discriminant is determined by a discriminant_association in a subtype_indication, the accessibility level of the object or subprogram designated by the associated value (or library level if the value is null);

12.a/2 **Discussion:** This deals with the following cases, when they occur in the context of an allocator or return statement:

12.b/2 • An extension_aggregate where the ancestor_part is a subtype_mark denoting a constrained subtype;

12.c/2 • An uninitialized allocator where the subtype_indication defines a constrained subtype;

12.d/2 • A discriminant of an object with a constrained nominal subtype, including constrained components, the result of calling a function with a constrained result subtype, the dereference of an access-to-constrained subtype, etc.

12.e/3 **Ramification:** {*AI05-0281-1*} The subtype_indication mentioned in this bullet is not necessarily the one given in the allocator or return statement that is determining the accessibility level; the constrained subtype might have been defined in an earlier declaration (as a named subtype).

12.f/3 {*AI05-0005-1*} If the value for this rule and the next one is derived from an Unchecked_Access attribute, the accessibility is library-level no matter what the accessibility level of the object is (see 13.10).

12.2/3 • {*AI05-0234-1*} If the value of the access discriminant is determined by a default_expression in the declaration of the discriminant, the level of the object or subprogram designated by the associated value (or library level if null);

12.f.1/3 **Discussion:** This covers the case of an unconstrained subcomponent of a limited type with defaulted access discriminants.

12.3/3 • {*AI05-0004-1*} If the value of the access discriminant is determined by a record_component_association in an aggregate, the accessibility level of the object or subprogram designated by the associated value (or library level if the value is null);

Discussion: In this bullet, the aggregate has to occur in the context of an allocator or return statement, while the subtype_indication of the previous bullet can occur anywhere (it doesn't have to be directly given in the allocator or return statement). 12.g/2

- In other cases, where the value of the access discriminant is determined by an object with an unconstrained nominal subtype, the accessibility level of the object. 12.4/3

  Discussion: {*AI95-00416-01*} In other words, if you know the value of the discriminant for an allocator or return statement from a discriminant constraint or an aggregate component association, then that determines the accessibility level; if you don't know it, then it is based on the object itself. 12.h/2

- {*AI95-00416-01*} The accessibility level of the anonymous access type of an access discriminant in any other context is that of the enclosing object. 12.5/3

- {*AI95-00162-01*} {*AI95-00254-01*} {*AI05-0270-1*} The accessibility level of the anonymous access type of an access parameter specifying an access-to-object type is the same as that of the view designated by the actual (or library-level if the actual is null). 13/3

  Ramification: {*AI05-0005-1*} If the value of the actual is derived from an Unchecked_Access attribute, the accessibility is always library-level (see 13.10). 13.a/3

- {*AI95-00254-01*} The accessibility level of the anonymous access type of an access parameter specifying an access-to-subprogram type is deeper than that of any master; all such anonymous access types have this same level. 13.1/2

  Reason: These represent "downward closures" and thus require passing of static links or global display information (along with generic sharing information if the implementation does sharing) along with the address of the subprogram. We must prevent conversions of these to types with "normal" accessibility, as those typically don't include the extra information needed to make a call. 13.b/2

- {*AI12-0070-1*} The accessibility level of the anonymous access subtype defined by a return_subtype_indication that is an access_definition (see 6.5) is that of the result subtype of the enclosing function. 13.2/4

- {*AI05-0148-1*} {*AI05-0240-1*} {*AI12-0070-1*} The accessibility level of the type of a stand-alone object of an anonymous access-to-object type is the same as the accessibility level of the type of the access value most recently assigned to the object[; accessibility checks ensure that this is never deeper than that of the declaration of the stand-alone object]. 13.3/4

  Proof: {*AI12-0005-1*} Conversions *into* the anonymous access type of a stand-alone object use a stricter (static) accessibility rule - see 4.6; these checks are the ones referred to above. 13.c/5

- {*AI95-00416-01*} {*AI05-0051-1*} {*AI05-0253-1*} The accessibility level of an object created by an allocator is the same as that of the access type, except for an allocator of an anonymous access type (an *anonymous allocator*) in certain contexts, as follows: For an anonymous allocator that defines the result of a function with an access result, the accessibility level is determined as though the allocator were in place of the call of the function; in the special case of a call that is the operand of a type conversion, the level is that of the target access type of the conversion. For an anonymous allocator defining the value of an access parameter, the accessibility level is that of the innermost master of the call. For an anonymous allocator whose type is that of a stand-alone object of an anonymous access-to-object type, the accessibility level is that of the declaration of the stand-alone object. For one defining an access discriminant, the accessibility level is determined as follows: 14/3

  - {*AI95-00416-01*} {*AI05-0024-1*} for an allocator used to define the discriminant of an object, the level of the object; 14.1/3

  - {*AI95-00416-01*} {*AI05-0024-1*} for an allocator used to define the constraint in a subtype_indication in any other context, the level of the master that elaborates the subtype_indication. 14.2/3

  - *This paragraph was deleted.*{*AI95-00416-01*} {*AI05-0024-1*} 14.3/3

14.4/3 {*AI95-00416-01*} {*AI05-0024-1*} {*AI05-0066-1*} In the first case, the allocated object is said to be a *coextension* of the object whose discriminant designates it, as well as of any object of which the discriminated object is itself a coextension or subcomponent. If the allocated object is a coextension of an anonymous object representing the result of an aggregate or function call that is used (in its entirety) to directly initialize a part of an object, after the result is assigned, the coextension becomes a coextension of the object being initialized and is no longer considered a coextension of the anonymous object. All coextensions of an object [(which have not thus been transfered by such an initialization)] are finalized when the object is finalized (see 7.6.1).

14.a.1/2 **Ramification:** The rules of access discriminants are such that when the space for an object with a coextension is reclaimed, the space for the coextensions can be reclaimed. Hence, there is implementation advice (see 13.11) that an object and its coextensions all be allocated from the same storage pool (or stack frame, in the case of a declared object).

14.5/3 - {*AI05-0051-1*} Within a return statement, the accessibility level of the anonymous access type of an access result is that of the master of the call.

15/3 - {*AI05-0014-1*} The accessibility level of a view of an object or subprogram designated by an access value is the same as that of the access type.

15.a/3 **Discussion:** {*AI05-0005-1*} {*AI05-0014-1*} This rule applies even when no dereference exists, for example when an access value is passed as an access parameter. This rule ensures that implementations are not required to include dynamic accessibility values with all access values.

16 - The accessibility level of a component, protected subprogram, or entry of (a view of) a composite object is the same as that of (the view of) the composite object.

16.1/5 {*AI95-00416-01*} {*AI05-0262-1*} {*AI12-0236-1*} {*AI12-0317-1*} In the above rules, the operative constituents of a name or expression (see 4.4) are considered to be used in a given context if the enclosing name or expression is used in that context.

16.a/5 **Discussion:** This means that constructs like parenthesized expressions, qualified_expressions, and conditional_expressions are ignored for the purposes of calculating accessibility levels, determining the master of a function call, and so on.

17 One accessibility level is defined to be *statically deeper* than another in the following cases:

18/5 - {*AI12-0406-1*} For a master construct that is statically nested within another master construct, the accessibility level of the inner master construct is statically deeper than that of the outer master construct.

18.a/5 *This paragraph was deleted.*{*AI12-0406-1*}

18.b **To be honest:** If a given accessibility level is statically deeper than another, then each level defined to be the same as the given level is statically deeper than each level defined to be the same as the other level.

18.1/2 - {*AI95-00254-01*} The accessibility level of the anonymous access type of an access parameter specifying an access-to-subprogram type is statically deeper than that of any master; all such anonymous access types have this same level.

18.c/2 **Ramification:** This rule means that it is illegal to convert an access parameter specifying an access to subprogram to a named access to subprogram type, but it is allowed to pass such an access parameter to another access parameter (the implicit conversion's accessibility will succeed).

19/5 - {*AI95-00254-01*} {*AI05-0082-1*} {*AI12-0406-1*} The statically deeper relationship does not apply to the accessibility level of the following:

19.1/5 - {*AI12-0406-1*} the anonymous type of an access parameter specifying an access-to-object type;

19.2/5 - {*AI12-0406-1*} the type of a stand-alone object of an anonymous access-to-object type;

19.3/5 - {*AI12-0392-1*} {*AI12-0406-1*} a raise_expression;

19.4/5 - {*AI12-0406-1*} a descendant of a generic formal type;

- {*AI12-0406-1*} a descendant of a type declared in a generic formal package. 19.5/5

- {*AI05-0148-1*} {*AI12-0406-1*} When the statically deeper relationship does not apply, the accessibility level is not considered to be statically deeper, nor statically shallower, than any other. 19.6/5

  Ramification: {*AI12-0005-1*} In these cases, no static accessibility checks are made, and we use dynamic accessibility checks to prevent problems. 19.a/5

- *This paragraph was deleted.*{*AI05-0142-4*} {*AI05-0235-1*} {*AI12-0089-1*} {*AI12-0157-1*} {*AI12-0277-1*} {*AI12-0324-1*} {*AI12-0345-1*} 19.7/5

- {*AI05-0051-1*} {*AI05-0234-1*} {*AI05-0235-1*} {*AI12-0089-1*} {*AI12-0157-1*} {*AI12-0372-1*} When within a function body or the return expression of an expression function, the accessibility level of the master representing an execution of the function is statically deeper than that of the master of the function call invoking that execution[, independent of how the master of the function call is determined (see above)]. 19.8/5

  This paragraph was deleted.{*AI05-0235-1*} {*AI12-0345-1*} 19.b/5

  To be honest: {*AI12-0005-1*} "Function body" includes bodies of generic functions. 19.c/5

- {*AI12-0445-1*} [For determining whether one level is statically deeper than another when within a generic package body, the generic package is presumed to be instantiated at the same level as where it was declared; runtime checks are required in the case of more deeply nested instantiations.] 20/5

  Proof: {*AI05-0082-1*} A generic package does not introduce a new master, so it has the static level of its declaration; the rest follows from the other "statically deeper" rules. 20.a/3

- For determining whether one level is statically deeper than another when within the declarative region of a type_declaration, the current instance of the type is presumed to be an object created at a deeper level than that of the type. 21

  Ramification: In other words, the rules are checked at compile time of the type_declaration, in an assume-the-worst manner. 21.a

{*AI12-0345-1*} Notwithstanding other rules given above, the accessibility level of an entity that is tied to that of an explicitly aliased formal parameter of an enclosing function is considered (both statically and dynamically) to be the same as that of an entity whose accessibility level is tied to that of the return object of that function. 21.1/5

Ramification: This rule only applies when the level of an explicitly aliased parameter of a function is compared to that of the return object of the function. If a value designating the explicitly aliased parameter is created and stored in a stand-alone object or passed as a parameter, this special property is lost (even for the dynamic accessibility of anonymous access types in these contexts). 21.b/5

Implementation Note: When this rule applies, no dynamic accessibility check should be made, even when one would normally be part of the execution of the construct. All places where this rule applies are known at compile-time. 21.c/5

The accessibility level of all library units is called the *library level*; a library-level declaration or entity is one whose accessibility level is the library level. 22

Ramification: Library_unit_declarations are library level. Nested declarations are library level if they are nested only within packages (possibly more than one), and not within subprograms, tasks, etc. 22.a

To be honest: The definition of the accessibility level of the anonymous type of an access parameter specifying an access-to-object type cheats a bit, since it refers to the view designated by the actual, but access values designate objects, not views of objects. What we really mean is the view that "would be" denoted by an expression "X.**all**", where X is the actual, even though such an expression is a figment of our imagination. The definition is intended to be equivalent to the following more verbose version: The accessibility level of the anonymous type of an access parameter is as follows: 22.b/2

- if the actual is an expression of a named access type — the accessibility level of that type; 22.c
- if the actual is an allocator — the accessibility level of the execution of the called subprogram; 22.d
- if the actual is a reference to the Access attribute — the accessibility level of the view denoted by the prefix; 22.e/1

22.f
- if the actual is a reference to the Unchecked_Access attribute — library accessibility level;

22.g
- if the actual is an access parameter — the accessibility level of its type.

22.h Note that the allocator case is explicitly mentioned in the RM95, because otherwise the definition would be circular: the level of the anonymous type is that of the view designated by the actual, which is that of the access type.

22.i **Discussion:** A deeper accessibility level implies a shorter maximum lifetime. Hence, when a rule requires X to have a level that is "not deeper than" Y's level, this requires that X has a lifetime at least as long as Y. (We say "maximum lifetime" here, because the accessibility level really represents an upper bound on the lifetime; an object created by an allocator can have its lifetime prematurely ended by an instance of Unchecked_Deallocation.)

22.j Package elaborations are not masters, and are therefore invisible to the accessibility rules: an object declared immediately within a package has the same accessibility level as an object declared immediately within the declarative region containing the package. This is true even in the body of a package; it jibes with the fact that objects declared in a package_body live as long as objects declared outside the package, even though the body objects are not visible outside the package.

22.k Note that the level of the *view* denoted by X.**all** can be different from the level of the *object* denoted by X.**all**. The former is determined by the type of X; the latter is determined either by the type of the allocator, or by the master in which the object was declared. The former is used in several Legality Rules and runtime checks; the latter is used to define when X.**all** gets finalized. The level of a view reflects what we can conservatively "know" about the object of that view; for example, due to type_conversions, an access value might designate an object that was allocated by an allocator for a different access type.

22.l Similarly, the level of the view denoted by X.**all**.Comp can be different from the level of the object denoted by X.**all**.Comp.

22.m If Y is statically deeper than X, this implies that Y will be (dynamically) deeper than X in all possible executions.

22.n Most accessibility checking is done at compile time; the rules are stated in terms of "statically deeper than". The exceptions are:

22.o/2
- Checks involving access parameters of an access-to-object type. The fact that "statically deeper than" is not defined for the anonymous access type of an access parameter implies that any rule saying "shall not be statically deeper than" does not apply to such a type, nor to anything defined to have "the same" level as such a type.

22.o.1/3
- {*AI05-0082-1*} Checks involving generic formal types and their descendants. This is because the actual type can be more or less deeply nested than the generic unit. Note that this only applies to the generic unit itself, and not to the instance. Any static checks needed in the instance will be performed. Any other checks (such as those in the generic body) will require a run-time check of some sort (although implementations that macro-expand generics can determine the result of the check when the generic is expanded).

22.p/3
- {*AI05-0082-1*} Checks involving other entities and views within generic packages. This is because an instantiation can be at a level that is more deeply nested than the generic package itself. In implementations that use a macro-expansion model of generics, these violations can be detected at macro-expansion time. For implementations that share generics, run-time code is needed to detect the error.

22.q/2
- {*AI95-00318-02*} {*AI95-00344-01*} {*AI95-00416-01*} Checks during function return and allocators, for nested type extensions and access discriminants.

22.r/3 {*AI05-0005-1*} Note that runtime checks are not required for access discriminants (except during function returns and allocators), because their accessibility is determined statically by the accessibility level of the enclosing object.

22.s/2 The accessibility level of the result object of a function reflects the time when that object will be finalized; we don't allow pointers to the object to survive beyond that time.

22.t We sometimes use the terms "accessible" and "inaccessible" to mean that something has an accessibility level that is not deeper, or deeper, respectively, than something else.

22.u/2 **Implementation Note:** {*AI95-00318-02*} {*AI95-00344-01*} {*AI95-00416-01*} If an accessibility Legality Rule is satisfied, then the corresponding runtime check (if any) cannot fail (and a reasonable implementation will not generate any checking code) unless one of the cases requiring runtime checks mentioned previously is involved.

22.v Accessibility levels are defined in terms of the relations "the same as" and "deeper than". To make the discussion more concrete, we can assign actual numbers to each level. Here, we assume that library-level accessibility is level 0, and each level defined as "deeper than" is one level deeper. Thus, a subprogram directly called from the environment task (such as the main subprogram) would be at level 1, and so on.

22.w/2 Accessibility is not enforced at compile time for access parameters of an access-to-object type. The "obvious" implementation of the runtime checks would be inefficient, and would involve distributed overhead; therefore, an efficient method is given below. The "obvious" implementation would be to pass the level of the caller at each subprogram call, task creation, etc. This level would be incremented by 1 for each dynamically nested master. An

Accessibility_Check would be implemented as a simple comparison — checking that X is not deeper than Y would involve checking that X <= Y.

A more efficient method is based on passing *static* nesting levels (within constructs that correspond at run time to masters — packages don't count). Whenever an access parameter is passed, an implicit extra parameter is passed with it. The extra parameter represents (in an indirect way) the accessibility level of the anonymous access type, and, therefore, the level of the view denoted by a dereference of the access parameter. This is analogous to the implicit "Constrained" bit associated with certain formal parameters of an unconstrained but definite composite subtype. In this method, we avoid distributed overhead: it is not necessary to pass any extra information to subprograms that have no access parameters. For anything other than an access parameter and its anonymous type, the static nesting level is known at compile time, and is defined analogously to the RM95 definition of accessibility level (e.g. derived access types get their nesting level from their parent). Checking "not deeper than" is a "<=" test on the levels. 22.x

For each access parameter of an access-to-object type, the static depth passed depends on the actual, as follows: 22.y/2

- If the actual is an expression of a named access type, pass the static nesting level of that type. 22.z
- If the actual is an allocator, pass the static nesting level of the caller, plus one. 22.aa
- If the actual is a reference to the Access attribute, pass the level of the view denoted by the prefix. 22.bb/1
- If the actual is a reference to the Unchecked_Access attribute, pass 0 (the library accessibility level). 22.cc
- If the actual is an access parameter of an access-to-object type, usually just pass along the level passed in. However, if the static nesting level of the formal (access) parameter is greater than the static nesting level of the actual (access) parameter, the level to be passed is the minimum of the static nesting level of the access parameter and the actual level passed in. 22.dd/2

For the Accessibility_Check associated with a type_conversion of an access parameter of an access-to-object type of a given subprogram to a named access type, if the target type is statically nested within the subprogram, do nothing; the check can't fail in this case. Otherwise, check that the value passed in is <= the static nesting depth of the target type. The other Accessibility_Checks are handled in a similar manner. 22.ee/2

This method, using statically known values most of the time, is efficient, and, more importantly, avoids distributed overhead. 22.ff

{*AI05-0148-1*} The implementation of accessibility checks for stand-alone objects of anonymous access-to-object types can be similar to that for anonymous access-to-object parameters. A static level suffices; it can be calculated using rules similar to those previously described for access parameters. 22.ff.1/3

{*AI05-0148-1*} One important difference between the stand-alone access variables and access parameters is that one can assign a local access parameter to a more global stand-alone access variable. Similarly, one can assign a more global access parameter to a more local stand-alone access variable. 22.ff.2/3

{*AI05-0148-1*} For these cases, it is important to note that the "correct" static accessibility level for an access parameter assigned to a stand-alone access object is the minimum of the passed in level and the static accessibility level of the stand-alone object itself. This is true since the static accessibility level passed in might be deeper than that of the stand-alone object, but the dynamic accessibility of the passed in object clearly must be shallower than the stand-alone object (whatever is passed in must live at least as long as the subprogram call). We do not need to keep a more local static level as accesses to objects statically deeper than the stand-alone object cannot be stored into the stand-alone object. 22.ff.3/3

Discussion: Examples of accessibility: 22.gg

```
{AI05-0005-1} package body Lib_Unit is                                    22.hh/3
    type T is tagged ...;
    type A0 is access all T;
    Global: A0 := ...;
    procedure P(X: in out T) is
        Y: aliased T;
        type A1 is access all T;
        Ptr0: A0 := Global; -- OK.
        Ptr1: A1 := X'Access; -- OK.
    begin
        Ptr1 := Y'Access; -- OK;
        Ptr0 := A0(Ptr1); -- Illegal type conversion!
        Ptr0 := X'Access; -- Illegal reference to Access attribute!
        Ptr0 := Y'Access; -- Illegal reference to Access attribute!
        Global := Ptr0; -- OK.
    end P;
end Lib_Unit;
```

{*AI05-0005-1*} The above illegal statements are illegal because the accessibility levels of X and Y are statically deeper than the accessibility level of A0. In every possible execution of any program including this library unit, if P is called, the accessibility level of X will be (dynamically) deeper than that of A0. Note that the accessibility levels of X and Y are the same. 22.ii/3

22.jj/2 Here's an example involving access parameters of an access-to-object type:

22.kk
```
procedure Main is
    type Level_1_Type is access all Integer;
```

22.ll
```
    procedure P(X: access Integer) is
        type Nested_Type is access all Integer;
    begin
        ... Nested_Type(X) ... -- (1)
        ... Level_1_Type(X) ... -- (2)
    end P;
```

22.mm
```
    procedure Q(X: access Integer) is
        procedure Nested(X: access Integer) is
        begin
            P(X);
        end Nested;
    begin
        Nested(X);
    end Q;
```

22.nn
```
    procedure R is
        Level_2: aliased Integer;
    begin
        Q(Level_2'Access); -- (3)
    end R;
```

22.oo
```
    Level_1: aliased Integer;
begin
    Q(Level_1'Access); -- (4)
    R;
end Main;
```

22.pp The run-time Accessibility_Check at (1) can never fail, and no code should be generated to check it. The check at (2) will fail when called from (3), but not when called from (4).

22.qq Within a type_declaration, the rules are checked in an assume-the-worst manner. For example:

22.rr/3
```
{AI05-0298-1} package P is
    type Int_Ptr is access all Integer;
    type Rec(D: access Integer) is limited private;
private
    type Rec_Ptr is access all Rec;
    function F(X: Rec_Ptr) return Boolean;
    function G(X: access Rec) return Boolean;
    type Rec(D: access Integer) is
        limited record
            C1: Int_Ptr := Int_Ptr(D); -- Illegal!
            C2: Rec_Ptr := Rec'Access; -- Illegal!
            C3: Boolean := F(Rec'Access); -- Illegal!
            C4: Boolean := G(Rec'Access);
        end record;
end P;
```

22.ss C1, C2, and C3 are all illegal, because one might declare an object of type Rec at a more deeply nested place than the declaration of the type. C4 is legal, but the accessibility level of the object will be passed to function G, and constraint checks within G will prevent it from doing any evil deeds.

22.tt Note that we cannot defer the checks on C1, C2, and C3 until compile-time of the object creation, because that would cause violation of the privacy of private parts. Furthermore, the problems might occur within a task or protected body, which the compiler can't see while compiling an object creation.

23 The following attribute is defined for a prefix X that denotes an aliased view of an object:

24/1 X'Access {*8652/0010*} {*AI95-00127-01*} X'Access yields an access value that designates the object denoted by X. The type of X'Access is an access-to-object type, as determined by the expected type. The expected type shall be a general access type. X shall denote an aliased view of an object[, including possibly the current instance (see 8.6) of a limited type within its definition, or a formal parameter or generic formal object of a tagged type]. The view denoted by the prefix X shall satisfy the following additional requirements, presuming the expected type for X'Access is the general access type *A* with designated type *D*:

25
- If *A* is an access-to-variable type, then the view shall be a variable; [on the other hand, if *A* is an access-to-constant type, the view may be either a constant or a variable.]

Discussion: The current instance of a limited type is considered a variable. 25.a

- {*AI95-00363-01*} {*AI05-0008-1*} {*AI05-0041-1*} The view shall not be a subcomponent that depends on discriminants of an object unless the object is known to be constrained. 26/3

Discussion: This restriction is intended to be similar to the restriction on renaming discriminant-dependent subcomponents. 26.a

Reason: This prevents references to subcomponents that might disappear or move or change constraints after creating the reference. 26.b

Implementation Note: There was some thought to making this restriction more stringent, roughly: "X shall not denote a subcomponent of a variable with discriminant-dependent subcomponents, if the nominal subtype of the variable is an unconstrained definite subtype." This was because in some implementations, it is not just the discriminant-dependent subcomponents that might move as the result of an assignment that changed the discriminants of the enclosing object. However, it was decided not to make this change because a reasonable implementation strategy was identified to avoid such problems, as follows: 26.c

- Place nondiscriminant-dependent components with any aliased parts at offsets preceding any discriminant-dependent components in a discriminated record type with defaulted discriminants. 26.d
- Preallocate the maximum space for unconstrained discriminated variables with aliased subcomponents, rather than allocating the initial size and moving them to a larger (heap-resident) place if they grow as the result of an assignment. 26.e

Note that for objects of a by-reference type, it is not an error for a programmer to take advantage of the fact that such objects are passed by reference. Therefore, the above approach is also necessary for discriminated record types with components of a by-reference type. 26.f

To make the above strategy work, it is important that a component of a derived type is defined to be discriminant-dependent if it is inherited and the parent subtype constraint is defined in terms of a discriminant of the derived type (see 3.7). 26.g

- {*8652/0010*} {*AI95-00127-01*} {*AI95-00363-01*} If *A* is a named access type and *D* is a tagged type, then the type of the view shall be covered by *D*; if *A* is anonymous and *D* is tagged, then the type of the view shall be either *D*'Class or a type covered by *D*; if *D* is untagged, then the type of the view shall be *D*, and either: 27/2
  - {*AI95-00363-01*} the designated subtype of *A* shall statically match the nominal subtype of the view; or 27.1/2
  - {*AI95-00363-01*} {*AI05-0041-1*} {*AI12-0095-1*} *D* shall be discriminated in its full view and unconstrained in any partial view, and the designated subtype of *A* shall be unconstrained. 27.2/4

Implementation Note: This ensures that the dope for an aliased array object can always be stored contiguous with it, but need not be if its nominal subtype is constrained. 27.a

Ramification: {*8652/0010*} {*AI95-00127-01*} An access attribute can be used as the controlling operand in a dispatching call; see 3.9.2. 27.a.1/1

{*AI95-00363-01*} This does not require that types have a partial view in order to allow an access attribute of an unconstrained discriminated object, only that any partial view that does exist is unconstrained. 27.a.2/2

Discussion: {*AI12-0095-1*} We assume the worst in a generic body whether or not a formal subtype has a constrained partial view; specifically, in a generic body a discriminated subtype is considered to have a constrained partial view if it is a descendant of an untagged generic formal private or derived type (see 12.5.1 for the formal definition of this rule). 27.a.3/4

- {*AI05-0041-1*} The accessibility level of the view shall not be statically deeper than that of the access type *A*. 28/3

Ramification: In an instance body, a runtime check applies. 28.a

{*AI95-00230-01*} If *A* is an anonymous access-to-object type of an access parameter, then the view can never have a deeper accessibility level than *A*. The same is true for an anonymous access-to-object type of an access discriminant, except when X'Access is used to initialize an access discriminant of an object created by an allocator. The latter case is illegal if the accessibility level of X is statically deeper than that of the access type of the allocator; a runtime check is needed in the case where the initial value comes from an access parameter. Other anonymous access-to-object types have "normal" accessibility checks. 28.b/2

28.b.1/5 **Discussion:** {*AI12-0363-1*} Additional restrictions exist in the specialized needs annexes. For instance, C.6 includes additional restrictions on atomic and volatile prefixes of the Access attribute.

28.1/3 {*AI05-0041-1*} In addition to the places where Legality Rules normally apply (see 12.3), these requirements apply also in the private part of an instance of a generic unit.

29 A check is made that the accessibility level of X is not deeper than that of the access type *A*. If this check fails, Program_Error is raised.

29.a/2 **Ramification:** The check is needed for access parameters of an access-to-object type and in instance bodies.

29.a.1/3 {*AI05-0024-1*} Because there are no access parameters permitted for task entries, the accessibility levels are always comparable. We would have to switch to the terminology used in 4.8 and 6.5 based on inclusion within masters if we relax this restriction. That might introduce unacceptable distributed overhead.

29.b/3 **Implementation Note:** {*AI05-0148-1*} This check requires that some indication of lifetime is passed as an implicit parameter along with access parameters of an access-to-object type. A similar indication is required for stand-alone objects of anonymous access-to-object types.No such requirement applies to other anonymous access types, since the checks associated with them are all compile-time checks.

30/5 {*AI12-0439-1*} If the nominal subtype of X does not statically match the designated subtype of *A*, a view conversion of X to the designated subtype is evaluated (which can raise Constraint_Error — see 4.6) and the value of X'Access designates that view.

31 The following attribute is defined for a prefix P that denotes a subprogram:

32/5 P'Access {*AI95-00229-01*} {*AI95-00254-01*} {*AI05-0239-1*} {*AI12-0064-2*} P'Access yields an access value that designates the subprogram denoted by P. The type of P'Access is an access-to-subprogram type (*S*), as determined by the expected type. The accessibility level of P shall not be statically deeper than that of *S*. If *S* is nonblocking, P shall be nonblocking. In addition to the places where Legality Rules normally apply (see 12.3), these rules apply also in the private part of an instance of a generic unit. The profile of P shall be subtype conformant with the designated profile of *S*, and shall not be Intrinsic. If the subprogram denoted by P is declared within a generic unit, and the expression P'Access occurs within the body of that generic unit or within the body of a generic unit declared within the declarative region of the generic unit, then the ultimate ancestor of *S* shall be either a nonformal type declared within the generic unit or an anonymous access type of an access parameter.

32.a/2 **Discussion:** {*AI95-00229-01*} The part about generic bodies is worded in terms of the denoted subprogram, not the denoted view; this implies that renaming is invisible to this part of the rule. “Declared within the declarative region of the generic” is referring to child and nested generic units. This rule is partly to prevent contract model problems with respect to the accessibility rules, and partly to ease shared-generic-body implementations, in which a subprogram declared in an instance needs to have a different calling convention from other subprograms with the same profile.

32.b Overload resolution ensures only that the profile is type conformant. This rule specifies that subtype conformance is required (which also requires matching calling conventions). P cannot denote an entry because access-to-subprogram types never have the *entry* calling convention. P cannot denote an enumeration literal or an attribute function because these have intrinsic calling conventions.

Legality Rules

32.1/3 {*AI05-0188-1*} An expression is said to have *distributed accessibility* if it is

32.2/3 - a conditional_expression (see 4.5.7); or

32.3/5 - {*AI12-0236-1*} a declare_expression (see 4.5.9) whose *body*_expression has distributed accessibility; or

32.4/5 - a view conversion, qualified_expression, or parenthesized expression whose operand has distributed accessibility.

32.5/5 {*AI05-0188-1*} The statically deeper relationship does not apply to the accessibility level of an expression having distributed accessibility; that is, such an accessibility level is not considered to be statically deeper, nor statically shallower, than any other.

{*AI05-0188-1*} Any static accessibility requirement that is imposed on an expression that has distributed accessibility (or on its type) is instead imposed on the *dependent*_expressions of the underlying conditional_expression. This rule is applied recursively if a *dependent*_expression also has distributed accessibility. 32.6/5

Discussion: This means that any Legality Rule requiring that the accessibility level of an expression (or that of the type of an expression) shall or shall not be statically deeper than some other level also applies, in the case where the expression has distributed accessibility, to each *dependent*_expression of the underlying conditional_expression. 32.c/3

NOTE 1 The Unchecked_Access attribute yields the same result as the Access attribute for objects, but has fewer restrictions (see 13.10). There are other predefined operations that yield access values: an allocator can be used to create an object, and return an access value that designates it (see 4.8); evaluating the literal **null** yields a null access value that designates no entity at all (see 4.2). 33

NOTE 2 {*AI95-00230-01*} The predefined operations of an access type also include the assignment operation, qualification, and membership tests. Explicit conversion is allowed between general access types with matching designated subtypes; explicit conversion is allowed between access-to-subprogram types with subtype conformant profiles (see 4.6). Named access types have predefined equality operators; anonymous access types do not, but they can use the predefined equality operators for *universal_access* (see 4.5.2). 34/2

Reason: {*AI95-00230-01*} Anonymous access types can use the universal access equality operators declared in Standard, while named access types cannot for compatibility reasons. By not having equality operators for anonymous access types, we eliminate the need to specify exactly where the predefined operators for anonymous access types would be defined, as well as the need for an implementer to insert an implicit declaration for "=", etc. at the appropriate place in their symbol table. Note that ":=", 'Access, and ".**all**" are defined. 34.a/2

NOTE 3 The object or subprogram designated by an access value can be named with a dereference, either an explicit_dereference or an implicit_dereference. See 4.1. 35

NOTE 4 A call through the dereference of an access-to-subprogram value is never a dispatching call. 36

Proof: See 3.9.2. 36.a

NOTE 5 {*AI95-00254-01*} {*AI12-0440-1*} {*AI12-0451-1*} The Access attribute for subprograms and parameters of an anonymous access-to-subprogram type can be used together to implement “downward closures” — that is, to pass a more-nested subprogram as a parameter to a less-nested subprogram, as can be appropriate for an iterator abstraction or numerical integration. Downward closures can also be implemented using generic formal subprograms (see 12.6). Unlike for objects, there is no Unchecked_Access attribute for subprograms. 37/5

NOTE 6 {*AI12-0451-1*} Using an access-to-class-wide tagged type with a dispatching operation is a potentially more structured alternative to using an access-to-subprogram type. 38/5

NOTE 7 {*AI12-0442-1*} An implementation can consider two access-to-subprogram values to be unequal, even though they designate the same subprogram. For instance, this can happen because one points directly to the subprogram, while the other points to a special prologue that performs an Elaboration_Check and then jumps to the subprogram. See 4.5.2. 39/5

Ramification: If equality of access-to-subprogram values is important to the logic of a program, a reference to the Access attribute of a subprogram should be evaluated only once and stored in a global constant for subsequent use and equality comparison. 39.a

Examples

Example of use of the Access attribute: 40

{*AI12-0312-1*}
```
Becky : Person_Name := new Person(F);        -- see 3.10.1
Cars  : array (1..2) of aliased Car;
   ...
Becky.Vehicle := Cars(1)'Access;
Casey.Vehicle := Cars(2)'Access;
```
41/5

Extensions to Ada 83

We no longer make things like 'Last and ".component" (basic) operations of an access type that need to be "declared" somewhere. Instead, implicit dereference in a prefix takes care of them all. This means that there should never be a case when X.**all**'Last is legal while X'Last is not. See AI83-00154. 41.a

Incompatibilities With Ada 95

{*AI95-00363-01*} Aliased variables are not necessarily constrained in Ada 2005 (see 3.6). Therefore, a subcomponent of an aliased variable may disappear or change shape, and taking 'Access of such a subcomponent thus is illegal, while 41.b/2

the same operation would have been legal in Ada 95. Note that most allocated objects are still constrained by their initial value (see 4.8), and thus legality of 'Access didn't change for them. For example:

41.c/2
```
type T1 (D1 : Boolean := False) is
   record
      case D1 is
         when False =>
            C1 : aliased Integer;
         when True =>
            null;
      end case;
   end record;
type Acc_Int is access all Integer;
```

41.d/2
```
A_T : aliased T1;
Ptr : Acc_Int := A_T.C1'Access; -- Illegal in Ada 2005, legal in Ada 95
A_T := (D1 => True);            -- Raised Constraint_Error in Ada 95, but does not
                                -- in Ada 2005, so Ptr would become invalid when this
                                -- is assigned (thus Ptr is illegal).
```

41.e/2 {*AI95-00363-01*} If a discriminated full type has a partial view (private type) that is constrained, we do not allow 'Access on objects to create a value of an object of an access-to-unconstrained type. Ada 95 allowed this attribute and various access subtypes, requiring that the heap object be constrained and thus making details of the implementation of the private type visible to the client of the private type. See 4.8 for more on this topic.

41.f/2 {*AI95-00229-01*} {*AI95-00254-01*} **Amendment Correction:** Taking 'Access of a subprogram declared in a generic unit in the body of that generic is no longer allowed. Such references can easily be used to create dangling pointers, as Legality Rules are not rechecked in instance bodies. At the same time, the rules were loosened a bit where that is harmless, and also to allow any routine to be passed to an access parameter of an access-to-subprogram type. The now illegal uses of 'Access can almost always be moved to the private part of the generic unit, where they are still legal (and rechecked upon instantiation for possibly dangling pointers).

Extensions to Ada 95

41.g/2 {*8652/0010*} {*AI95-00127-01*} **Corrigendum:** Access attributes of objects of class-wide types can be used as the controlling parameter in a dispatching calls (see 3.9.2). This was an oversight in Ada 95.

41.h/2 {*AI95-00235-01*} **Amendment Correction:** The type of the prefix can now be used in resolving Access attributes. This allows more uses of the Access attribute to resolve. For example:

41.i/2
```
type Int_Ptr is access all Integer;
type Float_Ptr is access all Float;
```

41.j/2
```
function Zap (Val : Int_Ptr) return Float;
function Zap (Val : Float_Ptr) return Float;
```

41.k/2
```
Value : aliased Integer := 10;
```

41.l/2
```
Result1 : Float := Zap (Value'access); -- Ambiguous in Ada 95; resolves in Ada 2005.
Result2 : Float := Zap (Int_Ptr'(Value'access)); -- Resolves in Ada 95 and Ada 2005.
```

41.m/2 This change is upward compatible; any expression that does not resolve by the new rules would have failed a Legality Rule.

Wording Changes from Ada 95

41.n/2 {*AI95-00162-01*} Adjusted the wording to reflect the fact that expressions and function calls are masters.

41.o/2 {*AI95-00230-01*} {*AI95-00254-01*} {*AI95-00318-02*} {*AI95-00385-01*} {*AI95-00416-01*} Defined the accessibility of the various new kinds and uses of anonymous access types.

Incompatibilities With Ada 2005

41.p/3 {*AI05-0008-1*} **Correction:** Simplified the description of when a discriminant-dependent component is allowed as the prefix of 'Access to when the object is known to be constrained. This fixes a confusion as to whether a subcomponent of an object that is not certain to be constrained can be used as a prefix of 'Access. The fix introduces an incompatibility, as the rule did not apply in Ada 95 if the prefix was a constant; but it now applies no matter what kind of object is involved. The incompatibility is not too bad, since most kinds of constants are known to be constrained.

41.q/3 {*AI05-0041-1*} **Correction:** Corrected the checks for the constrainedness of the prefix of the Access attribute so that assume-the-worst is used in generic bodies. This may make some programs illegal, but those programs were at risk having objects disappear while valid access values still pointed at them.

Extensions to Ada 2005

{*AI05-0082-1*} **Correction:** Eliminated the static accessibility definition for generic formal types, as the actual can be more or less nested than the generic itself. This allows programs that were illegal for Ada 95 and for Ada 2005. 41.r/3

{*AI05-0148-1*} {*AI05-0253-1*} Eliminate the static accessibility definition for stand-alone objects of anonymous access-to-object types. This allows such objects to be used as temporaries without causing accessibility problems. 41.s/3

Wording Changes from Ada 2005

{*AI05-0014-1*} **Correction:** Corrected the rules so that the accessibility of the object designated by an access object is that of the access type, even when no dereference is given. The accessibility was not specified in the past. This correction applies to both Ada 95 and Ada 2005. 41.t/3

{*AI05-0024-1*} **Correction:** Corrected accessibility rules for access discriminants so that no cases are omitted. 41.u/3

{*AI05-0051-1*} {*AI05-0234-1*} {*AI05-0235-1*} {*AI05-0284-1*} **Correction:** Corrected accessibility rules for anonymous access return types and access discriminants in return statements. 41.v/3

{*AI05-0066-1*} **Correction:** Changed coextension rules so that coextensions that belong to an anonymous object are transfered to the ultimate object. 41.w/3

{*AI05-0142-4*} {*AI05-0188-1*} {*AI05-0235-1*} Defined the accessibility of explicitly aliased parameters (see 6.1) and conditional_expressions (see 4.5.7). 41.x/3

{*AI05-0234-1*} **Correction:** Defined the term “master of the call” to simplify other wording, especially that for the accessibility checks associated with return statements and explicitly aliased parameters. 41.y/3

{*AI05-0270-1*} **Correction:** Defined the (omitted) accessibility level of null values when those are passed as the actual of an access-to-object parameter. 41.z/3

Inconsistencies With Ada 2012

{*AI12-0278-1*} {*AI12-0390-1*} **Correction:** Defined that the accessibility of a function that returns an anonymous access type is the same for implicit and explicit conversions. This could cause code involving implicit conversions to named access types that is legal and does not raise an exception in original Ada 2012 to become illegal or raise Program_Error because of an accessibility failure in Ada 2022. This is more likely to prevent a dangling pointer bug than to prevent something useful. 41.aa/5

{*AI12-0345-1*} **Correction:** Tightened the cases where an explicitly aliased parameter has special accessibility, to avoid needing to pass the required dynamic accessibility to functions that have explicitly aliased parameters. The changes affects programs that use the dynamic accessibility of an explicitly aliased parameter within a return statement of a function (typically using anonymous access types). This can mean that a program that would have been legal and worked in Ada 2012 as defined would raise Program_Error or be rejected in Ada 2022. One such example is: 41.bb/5

```
type Rec is record                                                                  41.cc/5
    Comp : aliased Integer;
    ...
end record;

function F1 (A : aliased in out Rec) return access Integer is                       41.dd/5
   function F2 (B : access Integer) return access Integer
      is (B); -- (1)
begin
   return F2 (A.Comp'Access); -- (2)
end F1;
```

At (2), there is a check that the dynamic accessibility level of B is not deeper than the master of the call for F2 (which is defined to be the master of the call for F1). In Ada 2012, since the reference is inside of a return statement, the dynamic accessibility of A.Comp'Access is the master of the call for F1 - meaning the check at (2) should pass. In Ada 2022, the dynamic accessibility of A.Comp'Access is local for for F1 - meaning the check at (2) should fail and raise Program_Error. 41.ee/5

We're not aware of any Ada 2012 compilers that correctly implemented the rule as written (because of the overhead that is implied), so the practical effect of this change is to make the rules more consistent with actual practice. 41.ff/5

{*AI12-0345-1*} **Correction:** Adjusted the rules so that using a part of a return object in a return expression does not break the connection between the inner and outer function calls. This means that some cases that (unnecessarily) failed an accessibility check in Ada 2012 will not do so in Ada 2022. This change will mostly remove latent problems from Ada code; very little code depends on an accessibility check failing. 41.gg/5

Incompatibilities With Ada 2012

{*AI12-0027-1*} **Corrigendum:** Defined the accessibility of a value conversion, so that it is treated like an aggregate built at the point of the conversion. This was previously unspecified, so this change might be incompatible if an Ada 41.hh/4

implementation treated the accessibility of such conversions as that of the operand type; in that case, previous legal conversions probably will become illegal as the lifetime of the conversion is very short. However, code that could tell this difference is fairly rare (taking 'Access of a component of a result of a type conversion), code depending on this accessibility was not portable, and such code could have created an immediately dangling pointer if the conversion actually made a copy (which is necessary in some instances).

41.ii/5 {*AI12-0371-1*} The accessibility of a renaming of a formal subprogram in a generic package specification has changed to reflect the fact that a wrapper may be needed for the formal subprogram. Thus, 'Access of such a renaming found in an instance that is not library-level may become illegal. We think this case is unlikely to occur in practice, and a simple workaround is available (rename the actual at the point of the instance, use that rename as the prefix of 'Access).

Extensions to Ada 2012

41.jj/5 {*AI12-0402-1*} Adjusted the special accessibility for functions initializing objects to only apply to composite types. This makes some function calls of functions that return elementary types and have aliased parameters legal that otherwise would be illegal. Such a function cannot return a part of a parameter, and thus one does not need parameter checks to make that possible.

Wording Changes from Ada 2012

41.kk/4 {*AI12-0067-1*} **Corrigendum:** Corrected so that it is clear that explicitly aliased parameters in procedures have the same accessibility as other parameters. Only explicitly aliased parameters in functions are special.

41.ll/4 {*AI12-0070-1*} **Corrigendum:** The accessibility of an anonymous access type of an extended_return_statement is defined here rather than in 6.5 so that all accessibility rules are here.

41.mm/4 {*AI12-0089-1*} **Corrigendum:** Corrected a number of rules so that they clearly apply to generic functions as well as functions. (Remember, a generic function is not a function.)

41.nn/4 {*AI12-0095-1*} **Corrigendum:** Moved the assume the worst rule about constrainedness of the prefix of attribute Access to 12.5.1, as a number of places in the Reference Manual need this rule.

41.oo/4 {*AI12-0157-1*} **Corrigendum:** Ensured that the statically deeper relationship applies within the return expression of an expression function. (Dynamic properties apply by equivalence, but static properties are handled separately.)

41.pp/5 {*AI12-0064-2*} Added Nonblocking (see 9.5) matching to P'Access.

41.qq/5 {*AI12-0156-1*} Added text to define the accessibility of anonymous access types declaring a loop parameter.

41.rr/5 {*AI12-0236-1*} Added declare_expressions to the accessibility rules.

41.ss/5 {*AI12-0277-1*} **Correction:** Clarified the static level of explicitly aliased parameters.

41.tt/5 {*AI12-0372-1*} **Correction:** Redefined the statically deeper relationship for the master of a function call and locals of that call. This might cause programs to be rejected that were previously legal, but any such program should have failed a dynamic accessibility check. Thus this is not an incompatibility as it is changing the detection of an error from runtime to compile-time.

41.uu/5 {*AI12-0392-1*} **Correction:** Added wording to clarify that the accessibility of a raise_expression does not need any static checks (it is considered to match any accessibility required).

41.vv/5 {*AI12-0406-1*} **Correction:** Used the new term “master construct”, to put static accessibility rules on a firmer basis, including ensuring that those rules apply inside of generic bodies.

3.11 Declarative Parts

1 [A declarative_part contains declarative_items (possibly none).]

Syntax

2 declarative_part ::= {declarative_item}

3 declarative_item ::=
basic_declarative_item | body

4/1 {*8652/0009*} {*AI95-00137-01*} basic_declarative_item ::=
basic_declaration | aspect_clause | use_clause

5 body ::= proper_body | body_stub

proper_body ::= 6
  subprogram_body | package_body | task_body | protected_body

Static Semantics

{*AI95-00420-01*} The list of declarative_items of a declarative_part is called the *declaration list* of the declarative_part. 6.1/2

Dynamic Semantics

The elaboration of a declarative_part consists of the elaboration of the declarative_items, if any, in the order in which they are given in the declarative_part. 7

An elaborable construct is in the *elaborated* state after the normal completion of its elaboration. Prior to that, it is *not yet elaborated*. 8

Ramification: The elaborated state is only important for bodies; certain uses of a body raise an exception if the body is not yet elaborated. 8.a

Note that "prior" implies before the start of elaboration, as well as during elaboration. 8.b

The use of the term "normal completion" implies that if the elaboration propagates an exception or is aborted, the declaration is not elaborated. RM83 missed the aborted case. 8.c

For a construct that attempts to use a body, a check (Elaboration_Check) is performed, as follows: 9

- {*8652/0014*} {*AI95-00064-01*} For a call to a (non-protected) subprogram that has an explicit body, a check is made that the body is already elaborated. This check and the evaluations of any actual parameters of the call are done in an arbitrary order. 10/1

  Discussion: AI83-00180 specifies that there is no elaboration check for a subprogram defined by a pragma Interface (or equivalently, pragma Import). AI83-00430 specifies that there is no elaboration check for an enumeration literal. AI83-00406 specifies that the evaluation of parameters and the elaboration check occur in an arbitrary order. AI83-00406 applies to generic instantiation as well (see below). 10.a

  {*8652/0014*} {*AI95-00064-01*} {*AI05-0177-1*} A subprogram can be completed by a renaming-as-body, a null_procedure_declaration, or an expression_function_declaration, and we need to make an elaboration check on such a body, so we use “body” rather than subprogram_body above. 10.a.1/3

- {*AI05-0229-1*} For a call to a protected operation of a protected type (that has a body — no check is performed if the protected type is imported — see B.1), a check is made that the protected_body is already elaborated. This check and the evaluations of any actual parameters of the call are done in an arbitrary order. 11/3

  Discussion: A protected type has only one elaboration “bit”, rather than one for each operation, because one call may result in evaluating the barriers of other entries, and because there are no elaborable declarations between the bodies of the operations. In fact, the elaboration of a protected_body does not elaborate the enclosed bodies, since they are not considered independently elaborable. 11.a

  Note that there is no elaboration check when calling a task entry. Task entry calls are permitted even before the associated task_body has been seen. Such calls are simply queued until the task is activated and reaches a corresponding accept_statement. We considered a similar rule for protected entries — simply queuing all calls until the protected_body was seen, but felt it was not worth the possible implementation overhead, particularly given that there might be multiple instances of the protected type. 11.b

- For the activation of a task, a check is made by the activator that the task_body is already elaborated. If two or more tasks are being activated together (see 9.2), as the result of the elaboration of a declarative_part or the initialization for the object created by an allocator, this check is done for all of them before activating any of them. 12

  Reason: As specified by AI83-00149, the check is done by the activator, rather than by the task itself. If it were done by the task itself, it would be turned into a Tasking_Error in the activator, and the other tasks would still be activated. 12.a

- For the instantiation of a generic unit that has a body, a check is made that this body is already elaborated. This check and the evaluation of any explicit_generic_actual_parameters of the instantiation are done in an arbitrary order. 13

14 The exception Program_Error is raised if any of these checks fails.

Extensions to Ada 83

14.a/2 {*AI95-00114-01*} The syntax for declarative_part is modified to remove the ordering restrictions of Ada 83; that is, the distinction between basic_declarative_items and later_declarative_items within declarative_parts is removed. This means that things like use_clauses and object_declarations can be freely intermixed with things like bodies.

14.b The syntax rule for proper_body now allows a protected_body, and the rules for elaboration checks now cover calls on protected operations.

Wording Changes from Ada 83

14.c The syntax rule for later_declarative_item is removed; the syntax rule for declarative_item is new.

14.d RM83 defines "elaborated" and "not yet elaborated" for declarative_items here, and for other things in 3.1, "Declarations". That's no longer necessary, since these terms are fully defined in 3.1.

14.e In RM83, all uses of declarative_part are optional (except for the one in block_statement with a **declare**) which is sort of strange, since a declarative_part can be empty, according to the syntax. That is, declarative_parts are sort of "doubly optional". In Ada 95, these declarative_parts are always required (but can still be empty). To simplify description, we go further and say (see 5.6, "Block Statements") that a block_statement without an explicit declarative_part is equivalent to one with an empty one.

Wording Changes from Ada 95

14.f/2 {*8652/0009*} {*AI95-00137-01*} **Corrigendum:** Changed representation clauses to aspect clauses to reflect that they are used for more than just representation.

14.g/2 {*8652/0014*} {*AI95-00064-01*} **Corrigendum:** Clarified that the elaboration check applies to all kinds of subprogram bodies.

14.h/2 {*AI95-00420-01*} Defined "declaration list" to avoid confusion for various rules. Other kinds of declaration list are defined elsewhere.

3.11.1 Completions of Declarations

1/3 {*8652/0014*} {*AI95-00064-01*} {*AI05-0177-1*} Declarations sometimes come in two parts. A declaration that requires a second part is said to *require completion*. The second part is called the *completion* of the declaration (and of the entity declared), and is either another declaration, a body, or a pragma. A *body* is a body, an entry_body, a null_procedure_declaration or an expression_function_declaration that completes another declaration, or a renaming-as-body (see 8.5.4).

1.a **Discussion:** Throughout the RM95, there are rules about completions that define the following:

1.b • Which declarations require a corresponding completion.

1.c • Which constructs can only serve as the completion of a declaration.

1.d • Where the completion of a declaration is allowed to be.

1.e • What kinds of completions are allowed to correspond to each kind of declaration that allows one.

1.f Don't confuse this compile-time concept with the run-time concept of completion defined in 7.6.1.

1.g Note that the declaration of a private type (if limited) can be completed with the declaration of a task type, which is then completed with a body. Thus, a declaration can actually come in *three* parts.

1.h/3 {*AI95-00217-06*} {*AI05-0162-1*} An incomplete type (whether declared in the limited view of a package or not) may be completed by a private type declaration, so we can in fact have *four* parts.

1.i/3 {*AI05-0229-1*} In Ada 2012, there are no language-defined pragmas that act as completions. Pragma Import (which is obsolescent) has the effect of setting aspect Import to True; such an aspect makes giving a completion illegal. The wording that allows pragmas as completions was left as it is harmless and appears in many places in this Reference Manual.

Name Resolution Rules

2 A construct that can be a completion is interpreted as the completion of a prior declaration only if:

3 • The declaration and the completion occur immediately within the same declarative region;

- The defining name or defining_program_unit_name in the completion is the same as in the declaration, or in the case of a pragma, the pragma applies to the declaration; 4
- If the declaration is overloadable, then the completion either has a type-conformant profile, or is a pragma. 5

Legality Rules

{*AI05-0229-1*} An implicit declaration shall not have a completion. For any explicit declaration that is specified to *require completion*, there shall be a corresponding explicit completion, unless the declared entity is imported (see B.1). 6/3

To be honest: {*AI95-00217-06*} The implicit declarations occurring in a limited view do have a completion (the explicit declaration occurring in the full view) but that's a special case, since the implicit declarations are actually built from the explicit ones. So they do not *require* a completion, they have one by *fiat*. 6.a.1/2

Discussion: The implicit declarations of predefined operators are not allowed to have a completion. Enumeration literals, although they are subprograms, are not allowed to have a corresponding subprogram_body. That's because the completion rules are described in terms of constructs (subprogram_declarations) and not entities (subprograms). When a completion is required, it has to be explicit; the implicit null package_body that Clause 7 talks about cannot serve as the completion of a package_declaration if a completion is required. 6.a

At most one completion is allowed for a given declaration. Additional requirements on completions appear where each kind of completion is defined. 7

Ramification: A subunit is not a completion; the stub is. 7.a

If the completion of a declaration is also a declaration, then *that* declaration might have a completion, too. For example, a limited private type can be completed with a task type, which can then be completed with a task body. This is not a violation of the “at most one completion” rule. 7.b

A type is *completely defined* at a place that is after its full type definition (if it has one) and after all of its subcomponent types are completely defined. A type shall be completely defined before it is frozen (see 13.14 and 7.3). 8

Reason: Index types are always completely defined — no need to mention them. There is no way for a completely defined type to depend on the value of a (still) deferred constant. 8.a

NOTE 1 {*AI05-0229-1*} Completions are in principle allowed for any kind of explicit declaration. However, for some kinds of declaration, the only allowed completion is an implementation-defined pragma, and implementations are not required to have any such pragmas. 9/3

This paragraph was deleted.{*AI05-0229-1*} 9.a/3

NOTE 2 {*AI12-0449-1*} There are rules that prevent premature uses of declarations that have a corresponding completion. The Elaboration_Checks of 3.11 prevent such uses at run time for subprograms, protected operations, tasks, and generic units. The freezing rules (see 13.14) prevent, at compile time, premature uses of other entities such as private types and deferred constants. 10/5

Wording Changes from Ada 83

This subclause is new. It is intended to cover all kinds of completions of declarations, be they a body for a spec, a full type for an incomplete or private type, a full constant declaration for a deferred constant declaration, or a pragma Import for any kind of entity. 10.a

Wording Changes from Ada 95

{*8652/0014*} {*AI95-00064-01*} **Corrigendum:** Added a definition of *body*, which is different than body or **body**. 10.b/2

Wording Changes from Ada 2005

{*AI95-0177-1*} Added null procedures and expression functions that are completions to the definition of *body*. 10.c/3

4 Names and Expressions

{*AI05-0299-1*} [The rules applicable to the different forms of name and expression, and to their evaluation, are given in this clause.] 1/3

4.1 Names

[Names can denote declared entities, whether declared explicitly or implicitly (see 3.1). Names can also denote objects or subprograms designated by access values; the results of type_conversions or function_calls; subcomponents and slices of objects and values; protected subprograms, single entries, entry families, and entries in families of entries. Finally, names can denote attributes of any of the foregoing.] 1

Syntax

{*AI05-0003-1*} {*AI05-0139-2*} {*AI12-0125-3*} name ::= 2/5

direct_name | explicit_dereference
| indexed_component | slice
| selected_component | attribute_reference
| type_conversion | function_call
| character_literal | qualified_expression
| generalized_reference | generalized_indexing
| target_name

direct_name ::= identifier | operator_symbol 3

Discussion: {*AI95-00114-01*} character_literal is no longer a direct_name. character_literals are usable even when the corresponding enumeration type declaration is not visible. See 4.2. 3.a/2

prefix ::= name | implicit_dereference 4

explicit_dereference ::= name.**all** 5

implicit_dereference ::= name 6

{*AI05-0004-1*} [Certain forms of name (indexed_components, selected_components, slices, and attribute_references) include a prefix that is either itself a name that denotes some related entity, or an implicit_dereference of an access value that designates some related entity.] 7/3

Name Resolution Rules

The name in a *dereference* (either an implicit_dereference or an explicit_dereference) is expected to be of any access type. 8

Static Semantics

{*AI05-0008-1*} If the type of the name in a dereference is some access-to-object type *T*, then the dereference denotes a view of an object, the *nominal subtype* of the view being the designated subtype of *T*. If the designated subtype has unconstrained discriminants, the (actual) subtype of the view is constrained by the values of the discriminants of the designated object, except when there is a partial view of the type of the designated subtype that does not have discriminants, in which case the dereference is not constrained by its discriminant values. 9/3

Ramification: If the value of the name is the result of an access type conversion, the dereference denotes a view created as part of the conversion. The nominal subtype of the view is not necessarily the same as that used to create the designated object. See 4.6. 9.a

9.b **To be honest:** We sometimes refer to the nominal subtype of a particular kind of name rather than the nominal subtype of the view denoted by the name (presuming the name denotes a view of an object). These two uses of nominal subtype are intended to mean the same thing.

9.c/3 **Reason:** {*AI05-0008-1*} The last sentence was not present in Ada 95; it is necessary in Ada 2005 because general access types can designate unconstrained objects, which was not possible in Ada 95. Thus, the rules that had this effect in Ada 95 (the object being constrained by its initial value) don't work in Ada 2005 and we have to say this explicitly.

9.d/3 {*AI05-0008-1*} The "except" part of the last sentence prevents privacy "breaking", so that if a private type has discriminants only in the full view, they don't interfere with freely interassigning values between objects of the type, even when the objects live in the heap.

9.e/3 **Implementation Note:** {*AI05-0008-1*} Since we don't depend on whether the designated object is constrained, it is not necessary to include a constrained bit in every object that could be designated by a general access type.

10 If the type of the name in a dereference is some access-to-subprogram type *S*, then the dereference denotes a view of a subprogram, the *profile* of the view being the designated profile of *S*.

10.a **Ramification:** This means that the formal parameter names and default expressions to be used in a call whose name or prefix is a dereference are those of the designated profile, which need not be the same as those of the subprogram designated by the access value, since 'Access requires only subtype conformance, not full conformance.

Dynamic Semantics

11/2 {*AI95-00415-01*} The evaluation of a name determines the entity denoted by the name. This evaluation has no other effect for a name that is a direct_name or a character_literal.

12 [The evaluation of a name that has a prefix includes the evaluation of the prefix.] The evaluation of a prefix consists of the evaluation of the name or the implicit_dereference. The prefix denotes the entity denoted by the name or the implicit_dereference.

13 The evaluation of a dereference consists of the evaluation of the name and the determination of the object or subprogram that is designated by the value of the name. A check is made that the value of the name is not the null access value. Constraint_Error is raised if this check fails. The dereference denotes the object or subprogram designated by the value of the name.

Examples

14 *Examples of direct names:*

15
```
Pi       -- the direct name of a number              (see 3.3.2)
Limit    -- the direct name of a constant            (see 3.3.1)
Count    -- the direct name of a scalar variable     (see 3.3.1)
Board    -- the direct name of an array variable     (see 3.6.1)
Matrix   -- the direct name of a type                (see 3.6)
Random   -- the direct name of a function            (see 6.1)
Error    -- the direct name of an exception          (see 11.1)
```

16 *Examples of dereferences:*

17
```
Next_Car.all     -- explicit dereference denoting the object designated
                 -- by the access variable Next_Car (see 3.10.1)
Next_Car.Owner   -- selected component with implicit dereference;
                 -- same as Next_Car.all.Owner
```

Extensions to Ada 83

17.a Type conversions and function calls are now considered names that denote the result of the operation. In the case of a type conversion used as an actual parameter or that is of a tagged type, the type conversion is considered a variable if the operand is a variable. This simplifies the description of "parameters of the form of a type conversion" as well as better supporting an important OOP paradigm that requires the combination of a conversion from a class-wide type to some specific type followed immediately by component selection. Function calls are considered names so that a type conversion of a function call and the function call itself are treated equivalently in the grammar. A function call is considered the name of a constant, and can be used anywhere such a name is permitted. See 6.5.

Type conversions of a tagged type are permitted anywhere their operand is permitted. That is, if the operand is a variable, then the type conversion can appear on the left-hand side of an assignment_statement. If the operand is an object, then the type conversion can appear in an object renaming or as a prefix. See 4.6. 17.b/1

Wording Changes from Ada 83

{*AI95-00114-01*} Everything of the general syntactic form name(...) is now syntactically a name. In any realistic parser, this would be a necessity since distinguishing among the various name(...) constructs inevitably requires name resolution. In cases where the construct yields a value rather than an object, the name denotes a value rather than an object. Names already denote values in Ada 83 with named numbers, components of the result of a function call, etc. This is partly just a wording change, and partly an extension of functionality (see Extensions heading above). 17.c/2

The syntax rule for direct_name is new. It is used in places where direct visibility is required. It's kind of like Ada 83's simple_name, but simple_name applied to both direct visibility and visibility by selection, and furthermore, it didn't work right for operator_symbols. The syntax rule for simple_name is removed, since its use is covered by a combination of direct_name and selector_name. The syntactic categories direct_name and selector_name are similar; it's mainly the visibility rules that distinguish the two. The introduction of direct_name requires the insertion of one new explicit textual rule: to forbid statement_identifiers from being operator_symbols. This is the only case where the explicit rule is needed, because this is the only case where the declaration of the entity is implicit. For example, there is no need to syntactically forbid (say) “X: "Rem";”, because it is impossible to declare a type whose name is an operator_symbol in the first place. 17.d

{*AI12-0005-1*} The syntax rules for explicit_dereference and implicit_dereference are new; this makes other rules simpler, since dereferencing an access value has substantially different semantics from selected_components. We also use name instead of prefix in the explicit_dereference rule since that seems clearer. Note that these rules rely on the fact that function calls are now names, so we don't need to use prefix to allow function calls in front of .**all**. 17.e/3

Discussion: Actually, it would be reasonable to allow any primary in front of .**all**, since only the value is needed, but that would be a bit radical. 17.f

We no longer use the term *appropriate for a type* since we now describe the semantics of a prefix in terms of implicit dereference. 17.g

Extensions to Ada 2005

{*AI05-0003-1*} A qualified_expression is now a name denoting a constant view; this allows them to be used as a prefix and to be renamed as an object. They are often used to remove ambiguity from function calls, and there may be no other way to do that. Interestingly, a type_conversion of a qualified_expression is already legal in these contexts, so this change mainly reduces clutter by eliminating an otherwise unneeded type_conversion from some expressions. 17.h/3

Wording Changes from Ada 2005

{*AI05-0008-1*} **Correction:** Added a missing rule so that most dereferences are assumed constrained (without determining whether the designated object is). This is just confirming the Ada 95 rules; Ada 2005 failed to ensure that this property was unchanged. 17.i/3

{*AI05-0139-2*} {*AI05-0299-1*} Added generalized_reference and generalized_indexing as types of name; these are documented as extensions in the appropriate subclauses. 17.j/3

Wording Changes from Ada 2012

{*AI12-0125-3*} Added a target_name (see 5.2.1) to the syntax of name. 17.k/5

4.1.1 Indexed Components

[An indexed_component denotes either a component of an array or an entry in a family of entries.] 1

Syntax

indexed_component ::= prefix(expression {, expression}) 2

Name Resolution Rules

The prefix of an indexed_component with a given number of expressions shall resolve to denote an array (after any implicit dereference) with the corresponding number of index positions, or shall resolve to denote an entry family of a task or protected object (in which case there shall be only one expression). 3

4 The expected type for each expression is the corresponding index type.

Static Semantics

5 When the prefix denotes an array, the indexed_component denotes the component of the array with the specified index value(s). The nominal subtype of the indexed_component is the component subtype of the array type.

6 When the prefix denotes an entry family, the indexed_component denotes the individual entry of the entry family with the specified index value.

Dynamic Semantics

7 For the evaluation of an indexed_component, the prefix and the expressions are evaluated in an arbitrary order. The value of each expression is converted to the corresponding index type. A check is made that each index value belongs to the corresponding index range of the array or entry family denoted by the prefix. Constraint_Error is raised if this check fails.

Examples

8 *Examples of indexed components:*

9
```
My_Schedule(Sat)   -- a component of a one-dimensional array   (see 3.6.1)
Page(10)           -- a component of a one-dimensional array   (see 3.6)
Board(M, J + 1)    -- a component of a two-dimensional array   (see 3.6.1)
Page(10)(20)       -- a component of a component               (see 3.6)
Request(Medium)    -- an entry in a family of entries          (see 9.1)
Next_Frame(L)(M, N) -- a component of a function call          (see 6.1)
```

10/5 {*AI12-0452-1*} Distinct notations are used for components of multidimensional arrays (such as Board) and arrays of arrays (such as Page). The components of an array of arrays are arrays and can therefore be indexed. Thus Page(10)(20) denotes the 20th component of Page(10). In the last example Next_Frame(L) is a function call returning an access value that designates a two-dimensional array.

4.1.2 Slices

1 [A slice denotes a one-dimensional array formed by a sequence of consecutive components of a one-dimensional array. A slice of a variable is a variable; a slice of a constant is a constant;] a slice of a value is a value.

Syntax

2 slice ::= prefix(discrete_range)

Name Resolution Rules

3 The prefix of a slice shall resolve to denote a one-dimensional array (after any implicit dereference).

4 The expected type for the discrete_range of a slice is the index type of the array type.

Static Semantics

5 A slice denotes a one-dimensional array formed by the sequence of consecutive components of the array denoted by the prefix, corresponding to the range of values of the index given by the discrete_range.

6 The type of the slice is that of the prefix. Its bounds are those defined by the discrete_range.

Dynamic Semantics

For the evaluation of a slice, the prefix and the discrete_range are evaluated in an arbitrary order. If the slice is not a *null slice* (a slice where the discrete_range is a null range), then a check is made that the bounds of the discrete_range belong to the index range of the array denoted by the prefix. Constraint_Error is raised if this check fails. 7

NOTE 1 {*AI12-0447-1*} By the rules given in 3.10.2, a slice is illegal as the prefix of an Access attribute_reference, even if the components or the array as a whole are aliased. 8/5

Proof: Slices are not aliased, by 3.10, "Access Types". 8.a

Reason: This is to ease implementation of general-access-to-array. If slices were aliased, implementations would need to store array dope with the access values, which is not always desirable given access-to-incomplete types completed in a package body. 8.b

NOTE 2 For a one-dimensional array A, the slice A(N .. N) denotes an array that has only one component; its type is the type of A. On the other hand, A(N) denotes a component of the array A and has the corresponding component type. 9

Examples

Examples of slices: 10

```
Stars(1 .. 15)          -- a slice of 15 characters          (see 3.6.3)
Page(10 .. 10 + Size)   -- a slice of 1 + Size components    (see 3.6)
Page(L)(A .. B)         -- a slice of the array Page(L)      (see 3.6)
Stars(1 .. 0)           -- a null slice                      (see 3.6.3)
My_Schedule(Weekday)    -- bounds given by subtype           (see 3.6.1 and 3.5.1)
Stars(5 .. 15)(K)       -- same as Stars(K)                  (see 3.6.3)
                        -- provided that K is in 5 .. 15
```
11

4.1.3 Selected Components

[Selected_components are used to denote components (including discriminants), entries, entry families, and protected subprograms; they are also used as expanded names as described below.] 1

Syntax

selected_component ::= prefix . selector_name 2

selector_name ::= identifier | character_literal | operator_symbol 3

Name Resolution Rules

A selected_component is called an *expanded name* if, according to the visibility rules, at least one possible interpretation of its prefix denotes a package or an enclosing named construct (directly, not through a subprogram_renaming_declaration or generic_renaming_declaration). 4

Discussion: See AI83-00187. 4.a

A selected_component that is not an expanded name shall resolve to denote one of the following: 5

Ramification: If the prefix of a selected_component denotes an enclosing named construct, then the selected_component is interpreted only as an expanded name, even if the named construct is a function that could be called without parameters. 5.a

- A component [(including a discriminant)]: 6

  The prefix shall resolve to denote an object or value of some non-array composite type (after any implicit dereference). The selector_name shall resolve to denote a discriminant_specification of the type, or, unless the type is a protected type, a component_declaration of the type. The selected_component denotes the corresponding component of the object or value. 7

  Reason: {*AI05-0005-1*} The components of a protected object cannot be named except by an expanded name, even from within the corresponding protected body. The protected body cannot reference the private components of some 7.a/3

arbitrary object of the protected type; the protected body may reference components of the current instance only (by an expanded name or a direct_name).

7.b **Ramification:** Only the discriminants and components visible at the place of the selected_component can be selected, since a selector_name can only denote declarations that are visible (see 8.3).

8 • A single entry, an entry family, or a protected subprogram:

9 The prefix shall resolve to denote an object or value of some task or protected type (after any implicit dereference). The selector_name shall resolve to denote an entry_declaration or subprogram_declaration occurring (implicitly or explicitly) within the visible part of that type. The selected_component denotes the corresponding entry, entry family, or protected subprogram.

9.a **Reason:** This explicitly says "visible part" because even though the body has visibility on the private part, it cannot call the private operations of some arbitrary object of the task or protected type, only those of the current instance (and expanded name notation has to be used for that).

9.1/2 • {*AI95-00252-01*} {*AI95-00407-01*} A view of a subprogram whose first formal parameter is of a tagged type or is an access parameter whose designated type is tagged:

9.2/3 {*AI95-00252-01*} {*AI95-00407-01*} {*AI05-0090-1*} The prefix (after any implicit dereference) shall resolve to denote an object or value of a specific tagged type *T* or class-wide type *T*'Class. The selector_name shall resolve to denote a view of a subprogram declared immediately within the declarative region in which an ancestor of the type *T* is declared. The first formal parameter of the subprogram shall be of type *T*, or a class-wide type that covers *T*, or an access parameter designating one of these types. The designator of the subprogram shall not be the same as that of a component of the tagged type visible at the point of the selected_component. The subprogram shall not be an implicitly declared primitive operation of type *T* that overrides an inherited subprogram implemented by an entry or protected subprogram visible at the point of the selected_component. The selected_component denotes a view of this subprogram that omits the first formal parameter. This view is called a *prefixed view* of the subprogram, and the prefix of the selected_component (after any implicit dereference) is called the *prefix* of the prefixed view.

9.b/3 **Discussion:** {*AI05-0090-1*} The part of the rule that excludes a primitive overriding subprogram as a selector applies only to the wrapper subprogram that is implicitly declared to override a subprogram inherited from a synchronized interface that is implemented by an operation of a task or protected type (see 9.1 and 9.4). We don't want calls that use a prefixed view to be ambiguous between the wrapper subprogram and the implementing entry or protected operation. Note that it is illegal to declare an explicit primitive that has a prefixed view that is homographic with one of the type's operations, so in normal cases it isn't possible to have an ambiguity in a prefix call. However, a class-wide operation of an ancestor type that is declared in the same declaration list with the ancestor type is also considered, and that can still make a call ambiguous.

10 An expanded name shall resolve to denote a declaration that occurs immediately within a named declarative region, as follows:

11 • The prefix shall resolve to denote either a package [(including the current instance of a generic package, or a rename of a package)], or an enclosing named construct.

12 • The selector_name shall resolve to denote a declaration that occurs immediately within the declarative region of the package or enclosing construct [(the declaration shall be visible at the place of the expanded name — see 8.3)]. The expanded name denotes that declaration.

12.a **Ramification:** Hence, a library unit or subunit can use an expanded name to refer to the declarations within the private part of its parent unit, as well as to other children that have been mentioned in with_clauses.

13 • If the prefix does not denote a package, then it shall be a direct_name or an expanded name, and it shall resolve to denote a program unit (other than a package), the current instance of a type, a block_statement, a loop_statement, or an accept_statement (in the case of an accept_-statement or entry_body, no family index is allowed); the expanded name shall occur within the declarative region of this construct. Further, if this construct is a callable construct and the prefix denotes more than one such enclosing callable construct, then the expanded name is ambiguous, independently of the selector_name.

Legality Rules

{*AI95-00252-01*} {*AI95-00407-01*} {*AI12-0204-1*} {*AI12-0427-1*} For a prefixed view of a subprogram whose first formal parameter is an access parameter, the prefix shall be legal as the prefix of an attribute_reference with attribute_designator Access appearing as the first actual parameter in a call on the unprefixed view of the subprogram. 13.1/5

Ramification: {*AI12-0204-1*} This rule prevents, for instance, using a nonaliased prefix in such a prefixed view. It also prevents using discriminant-dependent components as the prefix of such a prefixed view if those components would not be allowed to be renamed. 13.a/5

{*AI95-00407-01*} For a subprogram whose first parameter is of mode **in out** or **out**, or of an anonymous access-to-variable type, the prefix of any prefixed view shall denote a variable. 13.2/2

Reason: We want calls through a prefixed view and through a normal view to have the same legality. Thus, the implicit 'Access in this new notation needs the same legality check that an explicit 'Access would have. Similarly, we need to prohibit the object from being constant if the first parameter of the subprogram is **in out**, because that is (obviously) prohibited for passing a normal parameter. 13.b/2

Dynamic Semantics

The evaluation of a selected_component includes the evaluation of the prefix. 14

For a selected_component that denotes a component of a variant, a check is made that the values of the discriminants are such that the value or object denoted by the prefix has this component. The exception Constraint_Error is raised if this check fails. 15

Examples

Examples of selected components: 16

```
{AI95-00252-01} {AI95-00407-01} {AI12-0178} Tomorrow.Month      -- a record component      17/5
                                                                   (see 3.8)
Next_Car.Owner       -- a record component                         (see 3.10.1)
Next_Car.Owner.Age   -- a record component                         (see 3.10.1)
                     -- the previous two lines involve implicit dereferences
Writer.Unit          -- a record component (a discriminant)        (see 3.8.1)
Min_Cell(H).Value    -- a record component of the result           (see 6.1)
                     -- of the function call Min_Cell(H)
Cashier.Append       -- a prefixed view of a procedure             (see 3.9.4)
Control.Seize        -- an entry of a protected object             (see 9.4)
Pool(K).Write        -- an entry of the task Pool(K)               (see 9.1)
```

Examples of expanded names: 18

```
Key_Manager."<"      -- an operator of the visible part of a package   (see 7.3.1)      19
Dot_Product.Sum      -- a variable declared in a function body         (see 6.1)
Buffer.Pool          -- a variable declared in a protected unit        (see 9.11)
Buffer.Read          -- an entry of a protected unit                   (see 9.11)
Swap.Temp            -- a variable declared in a block statement       (see 5.6)
Standard.Boolean     -- the name of a predefined type                  (see A.1)
```

Extensions to Ada 83

We now allow an expanded name to use a prefix that denotes a rename of a package, even if the selector is for an entity local to the body or private part of the package, so long as the entity is visible at the place of the reference. This eliminates a preexisting anomaly where references in a package body may refer to declarations of its visible part but not those of its private part or body when the prefix is a rename of the package. 19.a

Wording Changes from Ada 83

The syntax rule for selector_name is new. It is used in places where visibility, but not necessarily direct visibility, is required. See 4.1, “Names” for more information. 19.b

The description of dereferencing an access type has been moved to 4.1, “Names”; name.**all** is no longer considered a selected_component. 19.c

The rules have been restated to be consistent with our new terminology, to accommodate class-wide types, etc. 19.d

Extensions to Ada 95

19.e/2 {*AI95-00252-01*} The prefixed view notation for tagged objects is new. This provides a similar notation to that used in other popular languages, and also reduces the need for use_clauses. This is sometimes known as "distinguished receiver notation".

19.f/2 Given the following definitions for a tagged type T:

19.g/2
```
procedure Do_Something (Obj : in out T; Count : in Natural);
procedure Do_Something_Else (Obj : access T; Flag : in Boolean);
My_Object : aliased T;
```

19.h/2 the following calls are equivalent:

19.i/2
```
Do_Something (My_Object, Count => 10);
My_Object.Do_Something (Count => 10);
```

19.j/2 as are the following calls:

19.k/2
```
Do_Something_Else (My_Object'Access, Flag => True);
My_Object.Do_Something_Else (Flag => True);
```

Wording Changes from Ada 2005

19.l/3 {*AI05-0090-1*} **Correction:** Corrected the definition of a prefixed view to ignore the implicit subprograms declared for "implemented by" entries and protected subprograms.

Incompatibilities With Ada 2012

19.m/5 {*AI12-0204-1*} **Correction:** Added a rule to ensure that all reasons that the prefix of an Access attribute can be illegal are covered by the rule for the implicit Access attribute of a prefixed view. If the object is a subcomponent that depends on discriminants or fails a static accessibility check, Ada 2012 would have allowed the prefix while Ada 2022 would not. This violated the principle that a prefixed view and a normal call have the same semantics; practically, the code may not have worked anyway if a compiler implemented generalized indexing by code expansion into the canonical form. Thus, such code wasn't practically portable.

4.1.4 Attributes

1 [An *attribute* is a characteristic of an entity that can be queried via an attribute_reference or a range_-attribute_reference.]

1.a/5 **Term entry: attribute** — characteristic or property of an entity that can be queried, and in some cases specified

Syntax

2/5 {*AI12-0262-1*} attribute_reference ::=
  prefix'attribute_designator
 | reduction_attribute_reference

3/2 {*AI05-0004-1*} attribute_designator ::=
  identifier[(*static*_expression)]
 | Access | Delta | Digits | Mod

4 range_attribute_reference ::= prefix'range_attribute_designator

5 range_attribute_designator ::= Range[(*static*_expression)]

Name Resolution Rules

6/5 {*AI12-0242-1*} {*AI12-0262-1*} In an attribute_reference that is not a reduction_attribute_reference, if the attribute_designator is for an attribute defined for (at least some) objects of an access type, then the prefix is never interpreted as an implicit_dereference; otherwise (and for all range_attribute_references and reduction_attribute_references), if there is a prefix and the type of the name within the prefix is of an access type, the prefix is interpreted as an implicit_dereference. Similarly, if the attribute_designator is for an attribute defined for (at least some) functions, then the prefix is never interpreted as a parameterless function_call; otherwise (and for all range_attribute_references and

reduction_attribute_references), if there is a prefix and the prefix consists of a name that denotes a function, it is interpreted as a parameterless function_call.

Discussion: The first part of this rule is essentially a "preference" against implicit dereference, so that it is possible to ask for, say, 'Size of an access object, without automatically getting the size of the object designated by the access object. This rule applies to 'Access, 'Unchecked_Access, 'Size, and 'Address, and any other attributes that are defined for at least some access objects. 6.a

The second part of this rule implies that, for a parameterless function F, F'Address is the address of F, whereas F'Size is the size of the anonymous constant returned by F. 6.b

We normally talk in terms of expected type or profile for name resolution rules, but we don't do this for attributes because certain attributes are legal independent of the type or the profile of the prefix. 6.c/1

{*AI95-00114-01*} Other than the rules given above, the Name Resolution Rules for the prefix of each attribute are defined as Name Resolution Rules for that attribute. If no such rules are defined, then no context at all should be used when resolving the prefix. In particular, any knowledge about the kind of entities required must not be used for resolution unless that is required by Name Resolution Rules. This matters in obscure cases; for instance, given the following declarations: 6.d/2

```
function Get_It return Integer is ... -- (1)
function Get_It return Some_Record_Type is ... -- (2)
```
6.e/2

the following attribute_reference cannot be resolved and is illegal: 6.f/2

```
if Get_It'Valid then
```
6.g/2

{*AI05-0005-1*} even though the Valid attribute is only defined for objects of scalar types, and thus cannot be applied to the result of function (2). That information cannot be used to resolve the prefix. The same would be true if (2) had been a procedure; even though the procedure does not denote an object, the attribute_reference is still illegal. 6.h/3

The expression, if any, in an attribute_designator or range_attribute_designator is expected to be of any integer type. 7

Legality Rules

The expression, if any, in an attribute_designator or range_attribute_designator shall be static. 8

Static Semantics

{*AI05-0006-1*} {*AI12-0032-1*} {*AI12-0159-1*} An attribute_reference denotes a value, an object, a subprogram, or some other kind of program entity. Unless explicitly specified otherwise, for an attribute_reference that denotes a value or an object, if its type is scalar, then its nominal subtype is the base subtype of the type; if its type is tagged, its nominal subtype is the first subtype of the type; otherwise, its nominal subtype is a subtype of the type without any constraint, null_exclusion, or predicate. Similarly, unless explicitly specified otherwise, for an attribute_reference that denotes a function, when its result type is scalar, its result subtype is the base subtype of the type, when its result type is tagged, the result subtype is the first subtype of the type, and when the result type is some other type, the result subtype is a subtype of the type without any constraint, null_exclusion, or predicate. 9/4

Ramification: The attributes defined by the language are summarized in K.2. Implementations can define additional attributes. 9.a

Discussion: {*AI05-0006-1*} The nominal subtype is primarily a concern when an attribute_reference, or a call on an attribute_reference, is used as the expression of a case statement, due to the full coverage requirement based on the nominal subtype. For nondiscrete cases, we define the nominal subtype mainly for completeness. Implementations may specify otherwise for implementation-defined attribute functions. 9.b/3

The rule is written to match the meaning of the italicized *T* in the definition of attributes such as Input; see 4.5.1. 9.c/3

To be honest: {*AI05-0006-1*} We don't worry about the fact that “base subtype” is not explicitly defined for the universal types. Since it is not possible to constrain a universal numeric type, all subtypes are unconstrained, and hence can be considered base subtypes. The wording above could be altered to bypass this issue, but it doesn't seem necessary, since universal integer is handled specially in the rules for case expression full coverage, and we don't allow user-defined functions for attribute functions whose result type is universal. 9.d/3

10 [A range_attribute_reference X'Range(N) is equivalent to the range X'First(N) .. X'Last(N), except that the prefix is only evaluated once. Similarly, X'Range is equivalent to X'First .. X'Last, except that the prefix is only evaluated once.]

Dynamic Semantics

11/5 {*AI12-0262-1*} The evaluation of a range_attribute_reference or an attribute_reference that is not a reduction_attribute_reference consists of the evaluation of the prefix.[The evaluation of a reduction_attribute_reference is defined in 4.5.10.]

Implementation Permissions

12/5 {*8652/0015*} {*AI95-00093-01*} {*AI12-0362-2*} An implementation may provide implementation-defined attributes; the identifier for such an implementation-defined attribute shall differ from those of the language-defined attributes.

12.a **Implementation defined:** Implementation-defined attributes.

12.b **Ramification:** They cannot be reserved words because reserved words are not legal identifiers.

12.c The semantics of implementation-defined attributes, and any associated rules, are, of course, implementation defined. For example, the implementation defines whether a given implementation-defined attribute can be used in a static expression.

13/5 {*AI12-0362-2*} An implementation may extend the definition of a language-defined attribute by accepting uses of that attribute that would otherwise be illegal in the following cases:

14/5 - in order to support compatibility with a previous edition of of this Reference Manual; or

14.a/1 **Ramification:** {*8652/0015*} {*AI95-00093-01*} Implementations are allowed to support the Small attribute for floating types, as this was defined in Ada 83, even though the name would conflict with a language-defined attribute.

15/5 - in the case of a language-defined attribute whose prefix is required by this document to be a floating point subtype, an implementation may accept an attribute_reference whose prefix is a fixed point subtype[; the semantics of such an attribute_reference are implementation defined.]

16 NOTE 1 Attributes are defined throughout this document, and are summarized in K.2.

17/5 NOTE 2 {*AI95-00235*} {*AI12-0447-1*} By the general rules given above, there is no expected type or profile for the name in a prefix of an attribute_reference (or a range_attribute_reference), which means that no context can be used to resolve the name. However, by the rules given in 3.10.2 for the case of the Access attribute, the expected type for the attribute_reference will be a single access type, and the resolution of the name can make use of the designated type or profile of this access type.

17.a/2 **Proof:** {*AI95-00235*} In the general case, there is no "expected type" for the prefix of an attribute_reference. In the special case of 'Access, there is an "expected type" or "expected profile" for the prefix.

17.b **Reason:** 'Access is a special case, because without it, it would be very difficult to take 'Access of an overloaded subprogram.

Examples

18 *Examples of attributes:*

19
```
Color'First          -- minimum value of the enumeration type Color  (see 3.5.1)
Rainbow'Base'First   -- same as Color'First                          (see 3.5.1)
Real'Digits          -- precision of the type Real                   (see 3.5.7)
Board'Last(2)        -- upper bound of the second dimension of Board (see 3.6.1)
Board'Range(1)       -- index range of the first dimension of Board  (see 3.6.1)
Pool(K)'Terminated   -- True if task Pool(K) is terminated           (see 9.1)
Date'Size            -- number of bits for records of type Date      (see 3.8)
Message'Address      -- address of the record variable Message
                     --                                             (see 3.7.1)
```

Extensions to Ada 83

19.a We now uniformly treat X'Range as X'First..X'Last, allowing its use with scalar subtypes.

We allow any integer type in the *static*_expression of an attribute designator, not just a value of *universal_integer*. The preference rules ensure upward compatibility. 19.b

Wording Changes from Ada 83

We use the syntactic category attribute_reference rather than simply "attribute" to avoid confusing the name of something with the thing itself. 19.c

The syntax rule for attribute_reference now uses identifier instead of simple_name, because attribute identifiers are not required to follow the normal visibility rules. 19.d

We now separate attribute_reference from range_attribute_reference, and enumerate the reserved words that are legal attribute or range attribute designators. We do this because identifier no longer includes reserved words. 19.e

The Ada 95 name resolution rules are a bit more explicit than in Ada 83. The Ada 83 rule said that the "meaning of the prefix of an attribute must be determinable independently of the attribute designator and independently of the fact that it is the prefix of an attribute". That isn't quite right since the meaning even in Ada 83 embodies whether or not the prefix is interpreted as a parameterless function call, and in Ada 95, it also embodies whether or not the prefix is interpreted as an implicit_dereference. So the attribute designator does make a difference — just not much. 19.f

Note however that if the attribute designator is Access, it makes a big difference in the interpretation of the prefix (see 3.10.2). 19.g

Wording Changes from Ada 95

{*8652/0015*} {*AI95-00093-01*} **Corrigendum:** The wording was changed to allow implementations to continue to implement the Ada 83 Small attribute. This was always intended to be allowed. 19.h/2

{*AI95-00235-01*} The note about resolving prefixes of attributes was updated to reflect that the prefix of an Access attribute now has an expected type (see 3.10.2). 19.i/2

Wording Changes from Ada 2005

{*AI05-0006-1*} **Correction:** Defined the nominal subtype of an attribute_reference to close a minor language hole. 19.j/3

Wording Changes from Ada 2012

{*AI12-0032-1*} **Corrigendum:** Allowed overriding the nominal subtype of an attribute_reference for an object; that is used elsewhere in this standard. 19.k/4

{*AI12-0159-1*} **Corrigendum:** Added wording so it is clear that predicates don't apply to the result of an attribute. 19.l/4

{*AI12-0242-1*} {*AI12-0262-1*} Added reduction_attribute_reference and cleaned up the rules here to avoid trampling the definition of those in 4.5.10. 19.m/5

{*AI12-0362-2*} Added a permission to support fixed point versions of float attributes, such as the rounding attributes found in A.5.3. 19.n/5

4.1.5 User-Defined References

Static Semantics

{*AI05-0139-2*} Given a discriminated type *T*, the following type-related operational aspect may be specified: 1/3

Implicit_Dereference 2/3

This aspect is specified by a name that denotes an access discriminant declared for the type *T*.

Aspect Description for Implicit_Dereference: Mechanism for user-defined implicit .**all**. 2.a/3

{*AI05-0139-2*} A (view of a) type with a specified Implicit_Dereference aspect is a *reference type*. A *reference object* is an object of a reference type. The discriminant named by the Implicit_Dereference aspect is the *reference discriminant* of the reference type or reference object. [A generalized_reference is a name that identifies a reference object, and denotes the object or subprogram designated by the reference discriminant of the reference object.] 3/3

Term entry: reference type — type that has user-defined behavior for ".all", defined by the Implicit_Dereference aspect 3.a/5

Syntax

4/3 {*AI05-0139-2*} generalized_reference ::= *reference_object*_name

Name Resolution Rules

5/3 {*AI05-0139-2*} {*AI05-0269-1*} The expected type for the *reference_object*_name in a generalized_reference is any reference type.

Static Semantics

5.1/4 {*AI12-0138-1*} The Implicit_Dereference aspect is nonoverridable (see 13.1.1).

5.a/4 **Reason:** This ensures that all descendants of a reference type have the same reference discriminant. This prevents generic contract problems with formal derived types.

6/3 {*AI05-0139-2*} A generalized_reference denotes a view equivalent to that of a dereference of the reference discriminant of the reference object.

7/5 {*AI05-0139-2*} {*AI12-0203-1*} Given a reference type *T*, the Implicit_Dereference aspect is inherited by descendants of type *T* if not overridden [(which is only permitted if confirming)]. If a descendant type constrains the value of the reference discriminant of *T* by a new discriminant, that new discriminant is the reference discriminant of the descendant. [If the descendant type constrains the value of the reference discriminant of *T* by an expression other than the name of a new discriminant, a generalized_reference that identifies an object of the descendant type denotes the object or subprogram designated by the value of this constraining expression.]

Dynamic Semantics

8/3 {*AI05-0139-2*} The evaluation of a generalized_reference consists of the evaluation of the *reference_object*_name and a determination of the object or subprogram designated by the reference discriminant of the named reference object. A check is made that the value of the reference discriminant is not the null access value. Constraint_Error is raised if this check fails. The generalized_reference denotes the object or subprogram designated by the value of the reference discriminant of the named reference object.

Examples

9/5 {*AI12-0429-1*} *Examples of the specification and use of generalized references:*

10/3
```
{AI05-0268-1} type Barrel is tagged ...  -- holds objects of type Element
```

11/3
```
{AI05-0139-2} {AI05-0299-1} type Ref_Element(Data : access Element) is limited
private
   with Implicit_Dereference => Data;
      -- This Ref_Element type is a "reference" type.
      -- "Data" is its reference discriminant.
```

12/3
```
{AI05-0139-2} {AI05-0268-1} function Find (B : aliased in out Barrel; Key : String)
   return Ref_Element;
   -- Returns a reference to an element of a barrel.
```

13/3
```
{AI05-0268-1} {AI05-0299-1} B: aliased Barrel;
```

14/3
```
{AI05-0139-2} ...
```

15/3
```
{AI05-0139-2} {AI05-0268-1} Find (B, "grape") := Element'(...);  -- Assign through a
reference.
```

16/3
```
{AI05-0139-2} {AI05-0268-1} -- This is equivalent to:
Find (B, "grape").Data.all := Element'(...);
```

Extensions to Ada 2005

16.a/3 {*AI05-0139-2*} The aspect Implicit_Dereference and the generalized_reference are new.

Incompatibilities With Ada 2012

{*AI12-0138-1*} **Corrigendum:** Defined Implicit_Dereference to be nonoverridable, which makes redefinitions and hiding of the aspect illegal. It's possible that some program could violate one of these new restrictions, but this is not very likely as reference types are not likely to be used in a hierarchy. 16.b/4

4.1.6 User-Defined Indexing

Static Semantics

{*AI05-0139-2*} Given a tagged type *T*, the following type-related, operational aspects may be specified: 1/3

{*AI12-0428-1*} Constant_Indexing 2/5

This aspect shall be specified by a name that denotes one or more functions declared immediately within the same declaration list in which *T*, or the declaration completed by *T*, is declared. All such functions shall have at least two parameters, the first of which is of type *T* or *T*'Class, or is an access-to-constant parameter with designated type *T* or *T*'Class.

Aspect Description for Constant_Indexing: Defines function(s) to implement user-defined indexed_components. 2.a/3

{*AI12-0428-1*} Variable_Indexing 3/5

This aspect shall be specified by a name that denotes one or more functions declared immediately within the same declaration list in which *T*, or the declaration completed by *T*, is declared. All such functions shall have at least two parameters, the first of which is of type *T* or *T*'Class, or is an access parameter with designated type *T* or *T*'Class. All such functions shall have a return type that is a reference type (see 4.1.5), whose reference discriminant is of an access-to-variable type.

Reason: We require these functions to return a reference type so that the object returned from the function can act like a variable. We need no similar rule for Constant_Indexing, since all functions return constant objects. 3.a/3

Aspect Description for Variable_Indexing: Defines function(s) to implement user-defined indexed_components. 3.b/3

{*AI12-0104-1*} These aspects are inherited by descendants of *T* (including the class-wide type *T*'Class). 4/4

Ramification: Indexing can be provided for multiple index types by overloading routines with different parameter profiles. For instance, the map containers provide indexing on both cursors and keys by providing pairs of overloaded routines to the Constant_Indexing and Variable_Indexing aspects. 4.a/3

{*AI05-0139-2*} {*AI05-0292-1*} An *indexable container type* is (a view of) a tagged type with at least one of the aspects Constant_Indexing or Variable_Indexing specified. An *indexable container object* is an object of an indexable container type. [A generalized_indexing is a name that denotes the result of calling a function named by a Constant_Indexing or Variable_Indexing aspect.] 5/3

Term entry: indexable container type — type that has user-defined behavior for indexing, via the Constant_Indexing or Variable_Indexing aspects 5.a/5

{*AI12-0138-1*} The Constant_Indexing and Variable_Indexing aspects are nonoverridable (see 13.1.1). 5.1/4

Reason: {*AI12-0160-1*} This (and the following Legality Rules) ensures that all descendants of an indexable container type have aspects with the same properties. This prevents generic contract problems with formal derived types. 5.b/5

{*AI12-0104-1*} {*AI12-0138-1*} A nonoverridable aspect allows the replacement of the implementation of an indexing function and the addition of a new indexing function for a derived type, but not the removal of an indexing function. This is necessary so that indexing can be used on objects of T'Class. So long as the tag of O is that of its nominal subtype, we do not want T'Class(O)(I) to mean something different than O(I). Thus we cannot allow a change in the function identified. As T'Class(O)(I) expands into a dispatching call, we need to ensure that there is a body for each such function -- but it is OK for that body to be changed from the original body (that's just normal dispatching). 5.c/4

Legality Rules

{*AI12-0160-1*} If an ancestor of a type *T* is an indexable container type, then any explicit specification of the Constant_Indexing or Variable_Indexing aspects shall be confirming; that is, the specified name shall match the inherited aspect (see 13.1.1). 6/5

Paragraphs 7 through 8 were deleted.

9/5 {*AI12-0160-1*} In addition to the places where Legality Rules normally apply (see 12.3), this rule applies also in the private part of an instance of a generic unit.

9.1/5 {*AI12-0204-1*} A generalized_indexing is illegal if the equivalent prefixed view (see below) is illegal.

Syntax

10/3 {*AI05-0139-2*} {*AI05-0292-1*} generalized_indexing ::= *indexable_container_object*_prefix actual_parameter_part

Name Resolution Rules

11/3 {*AI05-0139-2*} {*AI05-0292-1*} The expected type for the *indexable_container_object*_prefix of a generalized_indexing is any indexable container type.

11.a/5 **Ramification:** A prefix can be an implicit_dereference (see 4.1), so an access-to-*indexable_container_object* can be the prefix (English meaning!) of a generalized_indexing.

12/3 {*AI05-0139-2*} {*AI05-0292-1*} If the Constant_Indexing aspect is specified for the type of the *indexable_container_object*_prefix of a generalized_indexing, then the generalized_indexing is interpreted as a *constant indexing* under the following circumstances:

13/3 - when the Variable_Indexing aspect is not specified for the type of the *indexable_container_object*_prefix;

14/3 - when the *indexable_container_object*_prefix denotes a constant;

15/3 - when the generalized_indexing is used within a primary where a name denoting a constant is permitted.

15.a/3 **Ramification:** This means it is not interpreted as a constant indexing for the *variable*_name in the LHS of an assignment (not inside a primary), nor for the name used for an **out** or **in out** parameter (not allowed to be a constant), nor for the name in an object renaming (not inside a primary), unless there is no Variable_Indexing aspect defined.

16/3 Otherwise, the generalized_indexing is interpreted as a *variable indexing*.

17/3 When a generalized_indexing is interpreted as a constant (or variable) indexing, it is equivalent to a call on a prefixed view of one of the functions named by the Constant_Indexing (or Variable_Indexing) aspect of the type of the *indexable_container_object*_prefix with the given actual_parameter_part, and with the *indexable_container_object*_prefix as the prefix of the prefixed view.

17.a/3 **Ramification:** In other words, the generalized_indexing is equivalent to:

17.b/3 *indexable_container_object*_prefix.Indexing actual_parameter_part

17.c/4 {*AI12-0005-1*} where Indexing is the name specified for the Constant_Indexing or Variable_Indexing aspect. This equivalence is then resolved in the normal way; the aspect specifies a name, it does not denote declarations.

18/4 NOTE {*AI12-0104-1*} The Constant_Indexing and Variable_Indexing aspects cannot be redefined when inherited for a derived type, but the functions that they denote can be modified by overriding or overloading.

Examples

19/5 {*AI12-0429-1*} *Examples of the specification and use of generalized indexing:*

20/3
```
{AI05-0268-1} {AI05-0292-1} type Indexed_Barrel is tagged ...
  with Variable_Indexing => Find;
  -- Indexed_Barrel is an indexable container type,
  -- Find is the generalized indexing operation.
```

21/3
```
{AI05-0268-1} function Find (B : aliased in out Indexed_Barrel; Key : String)
   return Ref_Element;
   -- Return a reference to an element of a barrel (see 4.1.5).
```

22/3
```
{AI05-0268-1} IB: aliased Indexed_Barrel;
```

```
{AI05-0268-1} -- All of the following calls are then equivalent:                    23/3
Find (IB,"pear").Data.all := Element'(...); -- Traditional call
IB.Find ("pear").Data.all := Element'(...); -- Call of prefixed view
IB.Find ("pear")          := Element'(...); -- Implicit dereference (see 4.1.5)
IB      ("pear")          := Element'(...); -- Implicit indexing and dereference
IB      ("pear").Data.all := Element'(...); -- Implicit indexing only
```

Extensions to Ada 2005

{*AI05-0139-2*} Aspects Constant_Indexing and Variable_Indexing, and the generalized_indexing syntax are new. 23.a/3

Incompatibilities With Ada 2012

{*AI12-0160-1*} **Correction:** Prevented a derived type from specifying Constant_Indexing if the ancestor specified Variable_Indexing (and vice versa). This is necessary to preserve the intent that for an object Obj whose tag is that of its nominal subtype, T'Class(Obj)(I) always has the same meaning as Obj(I). Situations like this should be rare in practice; most types will either define both aspects or neither. 23.b/5

{*AI12-0204-1*} **Correction:** Added a rule that a generalized indexing is illegal if the equivalent prefixed view would be illegal. If the prefixed view would be illegal for any reason, Ada 2012 would have allowed the generalized indexing while Ada 2022 does not. This violated the principle that a generalized indexing and the equivalent prefixed view have the same semantics; practically, the code may not have worked anyway if a compiler implemented generalized indexing by code expansion into the canonical form. Thus, such code wasn't practically portable. 23.c/5

Wording Changes from Ada 2012

{*AI12-0104-1*} **Corrigendum:** Converted confusing and unnecessary normative wording about "overriding an aspect" into a note. 23.d/4

{*AI12-0138-1*} **Corrigendum:** Defined Constant_Indexing and Variable_Indexing to be nonoveridable. This is merely a new description for Legality Rules which already applied to these aspects. 23.e/4

{*AI12-0428-1*} **Correction:** Allowed the completion of a private type to use declarations from either the visible part or the private part. 23.f/5

4.2 Literals

[A *literal* represents a value literally, that is, by means of notation suited to its kind.] A literal is either a numeric_literal, a character_literal, the literal **null**, or a string_literal. 1

Discussion: An enumeration literal that is an identifier rather than a character_literal is not considered a *literal* in the above sense, because it involves no special notation “suited to its kind”. It might more properly be called an enumeration_identifier, except for historical reasons. 1.a

Name Resolution Rules

This paragraph was deleted.{*AI95-00230-01*} 2/2

For a name that consists of a character_literal, either its expected type shall be a single character type, in which case it is interpreted as a parameterless function_call that yields the corresponding value of the character type, or its expected profile shall correspond to a parameterless function with a character result type, in which case it is interpreted as the name of the corresponding parameterless function declared as part of the character type's definition (see 3.5.1). In either case, the character_literal denotes the enumeration_literal_specification. 3

Discussion: See 4.1.3 for the resolution rules for a selector_name that is a character_literal. 3.a

{*AI12-0325-1*} {*AI12-0373-1*} The expected type for a primary that is a string_literal shall be a single string type or a type with a specified String_Literal aspect (see 4.2.1). In either case, the string_literal is interpreted to be of its expected type. If the expected type of an integer literal is a type with a specified Integer_Literal aspect (see 4.2.1), the literal is interpreted to be of its expected type; otherwise it is interpreted to be of type *universal_integer*. If the expected type of a real literal is a type with a specified Real_Literal aspect (see 4.2.1), it is interpreted to be of its expected type; otherwise, it is interpreted to be of type *universal_real*. 4/5

Legality Rules

5 A character_literal that is a name shall correspond to a defining_character_literal of the expected type, or of the result type of the expected profile.

6/5 {*AI12-0295-1*} {*AI12-0325-1*} If the expected type for a string_literal is a string type, then for each character of the string_literal there shall be a corresponding defining_character_literal of the component type of the expected string type.

7/2 *This paragraph was deleted.*{*AI95-00230-01*} {*AI95-00231-01*}

Static Semantics

8/5 {*AI95-00230-01*} {*AI12-0373-1*} The literal **null** is of type *universal_access*.

Dynamic Semantics

9/5 {*AI12-0249-1*} If its expected type is a numeric type, the evaluation of a numeric literal yields the represented value. [In other cases, the effect of evaluating a numeric literal is determined by the Integer_Literal or Real_Literal aspect that applies (see 4.2.1).]

9.1/5 {*AI12-0249-1*} The evaluation of the literal **null** yields the null value of the expected type.

10/5 {*AI12-0295-1*} {*AI12-0325-1*} The evaluation of a string_literal that is a primary and has an expected type that is a string type, yields an array value containing the value of each character of the sequence of characters of the string_literal, as defined in 2.6. The bounds of this array value are determined according to the rules for positional_array_aggregates (see 4.3.3), except that for a null string literal, the upper bound is the predecessor of the lower bound. [In other cases, the effect of evaluating a string_literal is determined by the String_Literal aspect that applies (see 4.2.1).]

11/5 {*AI12-0295-1*} {*AI12-0325-1*} For the evaluation of a string_literal of a string type *T*, a check is made that the value of each character of the string_literal belongs to the component subtype of *T*. For the evaluation of a null string literal of a string type, a check is made that its lower bound is greater than the lower bound of the base range of the index type. The exception Constraint_Error is raised if either of these checks fails.

11.a/5 **Ramification:** {*AI12-0005-1*} If no predicates apply to the component subtype, the checks on the characters need not involve more than two checks altogether, since one need only check the characters of the string with the lowest and highest position numbers against the range of the component subtype.

12 NOTE Enumeration literals that are identifiers rather than character_literals follow the normal rules for identifiers when used in a name (see 4.1 and 4.1.3). Character_literals used as selector_names follow the normal rules for expanded names (see 4.1.3).

Examples

13 *Examples of literals:*

14
```
3.14159_26536    -- a real literal
1_345            -- an integer literal
'A'              -- a character literal
"Some Text"      -- a string literal
```

Incompatibilities With Ada 83

14.a Because character_literals are now treated like other literals, in that they are resolved using context rather than depending on direct visibility, additional qualification might be necessary when passing a character_literal to an overloaded subprogram.

Extensions to Ada 83

14.b Character_literals are now treated analogously to **null** and string_literals, in that they are resolved using context, rather than their content; the declaration of the corresponding defining_character_literal need not be directly visible.

Wording Changes from Ada 83

Name Resolution rules for enumeration literals that are not character_literals are not included anymore, since they are neither syntactically nor semantically "literals" but are rather names of parameterless functions. 14.c

Extensions to Ada 95

{*AI95-00230-01*} {*AI95-00231-01*} **Null** now has type *universal_access*, which is similar to other literals. **Null** can be used with anonymous access types. 14.d/2

Wording Changes from Ada 2012

{*AI12-0249-1*} {*AI12-0295-1*} The rules in this subclause are adjusted to allow for the possibility of user-defined literals. These are fully documented in the next subclause. 14.e/5

4.2.1 User-Defined Literals

{*AI12-0249-1*} Using one or more of the aspects defined below, a type may be specified to allow the use of one or more kinds of literals as values of the type. 1/5

Static Semantics

{*AI12-0249-1*} {*AI12-0342-1*} The following type-related operational aspects (collectively known as *user-defined literal aspects*) may be specified for a type *T*: 2/5

{*AI12-0249-1*} {*AI12-0342-1*} {*AI12-0373-1*} Integer_Literal 3/5

This aspect is specified by a *function*_name that statically denotes a function with a result type of *T* and one **in** parameter that is of type String and is not explicitly aliased.

Aspect Description for Integer_Literal: Defines a function to implement user-defined integer literals. 3.a/5

{*AI12-0249-1*} {*AI12-0342-1*} {*AI12-0373-1*} {*AI12-0394-1*} Real_Literal 4/5

This aspect is specified by a *function*_name that statically denotes a function with a result type of *T* and one **in** parameter that is of type String and is not explicitly aliased, and optionally a second function [(overloading the first)]with a result type of *T* and two **in** parameters of type String that are not explicitly aliased.

Aspect Description for Real_Literal: Defines a function or functions to implement user-defined real literals. 4.a/5

{*AI12-0295-1*} {*AI12-0342-1*} {*AI12-0373-1*} String_Literal 5/5

This aspect is specified by a *function*_name that statically denotes a function with a result type of *T* and one **in** parameter that is of type Wide_Wide_String and is not explicitly aliased.

Aspect Description for String_Literal: Defines a function to implement user-defined string literals. 5.a/5

Ramification: {*AI12-0342-1*} The following example is legal because the resolution of an aspect_definition for an aspect that is defined to be a subprogram is based on the profile required for that aspect (see 13.1.1): 5.b/5

```
package Pkg1 is                                                          5.c/5
   type T is record X, Y : Integer; end record
     with Integer_Literal => Int_Lit;
   function Int_Lit (X, Y : T) return Duration;     -- Wrong profile.
   function Int_Lit (Lit_Image : String) return T; -- Right profile.
end Pkg1;
```

{*AI12-0342-1*} {*AI12-0419-1*} User-defined literal aspects are nonoverridable (see 13.1.1). 6/5

Discussion: This means that in this example 6.a/5

```
package Pkg2 is                                                          6.b/5
   type T1 is record
      X, Y : Integer;
   end record with Integer_Literal => I_L;
   function I_L (S : String) return T1 is ((0, 0));                      6.c/5
   type T2 is new T1;                                                    6.d/5
   function I_L (S : String) return T2 is ((1, 1));
   X : T2 := 123;
end Pkg2;
```

6.e/5 {*AI12-0005-1*} {*AI12-0342-1*} {*AI12-0419-1*} the initial value of Pkg2.X is (1,1), not (0,0).

7/5 {*AI12-0342-1*} {*AI12-0427-1*} When a numeric literal is interpreted as a value of a non-numeric type *T* or a string_literal is interpreted as a value of a type *T* that is not a string type (see 4.2), it is equivalent to a call to the subprogram denoted by the corresponding aspect of *T*: the Integer_Literal aspect for an integer literal, the Real_Literal aspect for a real literal, and the String_Literal aspect for a string_literal. The actual parameter of this notional call is a string_literal representing a sequence of characters that is the same as the sequence of characters in the original numeric literal, or the sequence represented by the original string literal.

7.a/5 **Discussion:** This equivalence defines, for example, the nominal type, the nominal subtype, and the accessibility level of a user-defined literal. It also has the consequence that a user-defined literal shall not be of an abstract type (because that would be equivalent to a nondispatching call to an abstract function). This equivalence also defines the Dynamic Semantics of evaluating a user-defined literal.

7.b/5 The (sub)type of the actual parameter to this call is determined by the profile of the appropriate aspect, and the bounds of the string_literal are defined by the usual rules for the bounds of a string_literal.

8/5 {*AI12-0342-1*} Such a literal is said to be a *user-defined literal*.

9/5 {*AI12-0394-1*} When a named number that denotes a value of type *universal_integer* is interpreted as a value of a non-numeric type *T*, it is equivalent to a call to the function denoted by the Integer_Literal aspect of *T*. The actual parameter of this notional call is a String having a textual representation of a decimal integer literal optionally preceded by a minus sign, representing the same value as the named number.

10/5 {*AI12-0394-1*} When a named number that denotes a value of type *universal_real* is interpreted as a value of a non-numeric type *T*, it is equivalent to a call to the two-parameter function denoted by the Real_Literal aspect of *T*, if any. The actual parameters of this notional call are each a String with the textual representation of a decimal integer literal, with the first optionally preceded by a minus sign, where the first String represents the same value as the numerator, and the second the same value as the denominator, of the named number when represented as a rational number in lowest terms, with a positive denominator.

Legality Rules

11/5 {*AI12-0249-1*} {*AI12-0295-1*} {*AI12-0325-1*} {*AI12-0342-1*} The Integer_Literal or Real_Literal aspect shall not be specified for a type *T* if the full view of *T* is a numeric type. The String_Literal aspect shall not be specified for a type *T* if the full view of *T* is a string type.

12/5 {*AI12-0342-1*} For a nonabstract type, the function directly specified for a user-defined literal aspect shall not be abstract.

13/5 {*AI12-0342-1*} For a tagged type with a partial view, a user-defined literal aspect shall not be directly specified on the full type.

14/5 {*AI12-0342-1*} {*AI12-0394-1*} If a nonabstract tagged type inherits any user-defined literal aspect, then each inherited aspect shall be directly specified as a nonabstract function for the type unless the inherited aspect denotes a nonabstract function, or functions, and the type is a null extension.

15/5 {*AI12-0394-1*} If a named number that denotes a value of type *universal_integer* is interpreted as a value of a non-numeric type *T*, *T* shall have an Integer_Literal aspect. If a named number that denotes a value of type *universal_real* is interpreted as a value of a non-numeric type *T*, *T* shall have a Real_Literal aspect, and the aspect shall denote a function that has two **in** parameters, both of type String, with result of type *T*.

16/5 {*AI12-0249-1*} {*AI12-0342-1*} In addition to the places where Legality Rules normally apply (see 12.3), these rules also apply in the private part of an instance of a generic unit.

Bounded (Run-Time) Errors

{*AI12-0249-1*} {*AI12-0325-1*} {*AI12-0342-1*} {*AI12-0394-1*} It is a bounded error if the evaluation of a literal or named number that has an expected type with a specified user-defined literal aspect propagates an exception. Either Program_Error or the exception propagated by the evaluation is raised at the point of use of the value of the literal or named number. If it is recognized prior to run time that evaluation of such a literal or named number will inevitably (if executed) result in such a bounded error, then this may be reported as an error prior to run time. 17/5

Implementation Note: {*AI12-0249-1*} As always, an implementation may apply "as-if" optimizations (those that result in the same external effects in the absence of erroneous execution) to the function calls associated with user-defined literals. In particular, if the function associated with a user-defined literal aspect has a Global aspect that indicates no references to global variables, then a number of optimizations are available to the implementation: 17.a/5

- The implementation can evaluate a user-defined literal function at compile-time if it has access to the body of the function (for example, if it is inlined), and that body doesn't reference anything evaluated at runtime. If the compile-time evaluation results in an exception, this bounded error allows the compilation unit to be rejected. 17.b/5
- Implementations can use existing permissions (see 6.1.2) to avoid evaluating the function associated with a user-defined literal more than once for a particular literal value. This evaluation can be "hoisted" (done once per compilation unit during the elaboration of the unit) if the compiler can prove that the function doesn't depend on any constants or locals with a runtime value not yet elaborated. 17.c/5
- If the literal value is not needed by the execution of the program, the function call can be omitted even if it might have side-effects (again, see 6.1.2). 17.d/5

Examples

{*AI12-0429-1*} *Examples of the specification and use of user-defined literals:* 18/5

```
{AI12-0312-1} subtype Roman_Character is Wide_Wide_Character                                    19/5
   with Static_Predicate =>
      Roman_Character in 'I' | 'V' | 'X' | 'L' | 'C' | 'D' | 'M';

{AI12-0312-1} Max_Roman_Number : constant := 3_999;  -- MMMCMXCIX                              20/5

{AI12-0312-1} type Roman_Number is range 1 .. Max_Roman_Number                                  21/5
   with String_Literal => To_Roman_Number;

{AI12-0312-1} function To_Roman_Number (S : Wide_Wide_String) return Roman_Number               22/5
   with Pre => S'Length > 0 and then
      (for all Char of S => Char in Roman_Character);

{AI12-0312-1} {AI12-0386-1} function To_Roman_Number (S : Wide_Wide_String) return              23/5
Roman_Number is
   (declare
      R : constant array (Integer range <>) of Roman_Number :=
         (for D in S'Range => Roman_Digit'Enum_Rep
             (Roman_Digit'Wide_Wide_Value ('''' & S(D) & ''')));
                   -- See 3.5.2 and 13.4
    begin
      [for I in R'Range =>
         (if I < R'Last and then R(I) < R(I + 1) then -1 else 1) * R(I)]
            'Reduce("+", 0)
   );

{AI12-0312-1} X : Roman_Number := "III" * "IV" * "XII"; -- 144 (that is, CXLIV)                 24/5
```

Extensions to Ada 2012

{*AI12-0249-1*} {*AI12-0295-1*} {*AI12-0325-1*} {*AI12-0342-1*} The user-defined literal aspects Integer_Literal, Real_Literal, and String_Literal are new. 24.a/5

4.3 Aggregates

[An *aggregate* combines component values into a composite value of an array type, record type, or record extension.] 1

1.a/5 **Term entry: aggregate** — construct used to define a value of a composite type by specifying the values of the components of the type

Syntax

2/5 {*AI12-0127-1*} {*AI12-0212-1*} aggregate ::=
record_aggregate | extension_aggregate | array_aggregate
| delta_aggregate | container_aggregate

Name Resolution Rules

3/5 {*AI95-00287-01*} {*AI12-0127-1*} {*AI12-0212-1*} {*AI12-0307-1*} The expected type for an aggregate shall be a single array type, a single type with the Aggregate aspect specified, or a single descendant of a record type or of a record extension.

3.a **Discussion:** See 8.6, "The Context of Overload Resolution" for the meaning of "shall be a single ... type".

3.b/3 **Ramification:** {*AI05-0005-1*} There are additional rules for each kind of aggregate. These aggregate rules are additive; a legal expression needs to satisfy all of the applicable rules. That means the rule given here must be satisfied even when it is syntactically possible to tell which specific kind of aggregate is being used.

3.c/5 **Reason:** {*AI12-0005-1*} {*AI12-0127-1*} {*AI12-0212-1*} {*AI12-0307-1*} Generally, we don't want to look in aggregates to resolve them. For instance, Ada 95 allowed array types to match extension_aggregates, even though those have to be record types. Thus, we allow any record type with any visible (nondiscriminant) components to match an aggregate, even though only delta_aggregates allow private types or private extensions. Similarly, we allow any container type to match an aggregate, even though only container_aggregates allow container types.

Legality Rules

4/5 {*AI12-0127-1*} A record_aggregate or extension_aggregate shall not be of a class-wide type.

4.a **Ramification:** When the expected type in some context is class-wide, an aggregate has to be explicitly qualified by the specific type of value to be created, so that the expected type for the aggregate itself is specific.

4.b **Discussion:** We used to disallow aggregates of a type with unknown discriminants. However, that was unnecessarily restrictive in the case of an extension aggregate, and irrelevant to a record aggregate (since a type that is legal for a record aggregate could not possibly have unknown discriminants) and to an array aggregate (the only specific types that can have unknown discriminants are private types, private extensions, and types derived from them).

4.c/5 **Reason:** {*AI12-0127-1*} We don't mention delta_aggregates, as those can get the specific type from the object represented by the *base*_expression (possibly at run time). We don't mention array_aggregates, as those cannot even be of a tagged type, so being class-wide is impossible.

Dynamic Semantics

5 For the evaluation of an aggregate, an anonymous object is created and values for the components or ancestor part are obtained (as described in the subsequent subclause for each kind of the aggregate) and assigned into the corresponding components or ancestor part of the anonymous object. Obtaining the values and the assignments occur in an arbitrary order. The value of the aggregate is the value of this object.

5.a **Discussion:** The ancestor part is the set of components inherited from the ancestor type. The syntactic category ancestor_part is the expression or subtype_mark that specifies how the ancestor part of the anonymous object should be initialized.

5.b **Ramification:** The assignment operations do the necessary value adjustment, as described in 7.6. Note that the value as a whole is not adjusted — just the subcomponents (and ancestor part, if any). 7.6 also describes when this anonymous object is finalized.

5.c If the ancestor_part is a subtype_mark the Initialize procedure for the ancestor type is applied to the ancestor part after default-initializing it, unless the procedure is abstract, as described in 7.6. The Adjust procedure for the ancestor type is not called in this case, since there is no assignment to the ancestor part as a whole.

6 If an aggregate is of a tagged type, a check is made that its value belongs to the first subtype of the type. Constraint_Error is raised if this check fails.

Ramification: This check ensures that no values of a tagged type are ever outside the first subtype, as required for inherited dispatching operations to work properly (see 3.4). This check will always succeed if the first subtype is unconstrained. This check is not extended to untagged types to preserve upward compatibility. 6.a

Extensions to Ada 83

We now allow extension_aggregates. 6.b

Wording Changes from Ada 83

We have adopted new wording for expressing the rule that the type of an aggregate shall be determinable from the outside, though using the fact that it is nonlimited record (extension) or array. 6.c

An aggregate now creates an anonymous object. This is necessary so that controlled types will work (see 7.6). 6.d

Incompatibilities With Ada 95

{*AI95-00287-01*} In Ada 95, a limited type is not considered when resolving an aggregate. Since Ada 2005 now allows limited aggregates, we can have incompatibilities. For example: 6.e/2

```
type Lim is limited                                                    6.f/2
   record
      Comp: Integer;
   end record;

type Not_Lim is                                                        6.g/2
   record
      Comp: Integer;
   end record;

procedure P(X: Lim);                                                   6.h/2
procedure P(X: Not_Lim);

P((Comp => 123)); -- Illegal in Ada 2005, legal in Ada 95              6.i/2
```

The call to P is ambiguous in Ada 2005, while it would not be ambiguous in Ada 95 as the aggregate could not have a limited type. Qualifying the aggregate will eliminate any ambiguity. This construction would be rather confusing to a maintenance programmer, so it should be avoided, and thus we expect it to be rare. 6.j/2

Extensions to Ada 95

{*AI95-00287-01*} Aggregates can be of a limited type. 6.k/2

Incompatibilities With Ada 2012

{*AI12-0005-1*} {*AI12-0127-1*} {*AI12-0212-1*} We now allow types with the Aggregate aspect specified ("container types"), as well as private types and extensions descended from a record type or extension, to match all forms of aggregate. These types are only allowed for new types of aggregate (container_aggregates for the Aggregate aspect, and delta_aggregates for private types), but, consistent with other forms of aggregate, we do not look at the form of the aggregate to determine resolution. This can be incompatible in usually unlikely cases, where overloading of a container or private type with a type that was previously allowed in aggregates makes an existing call ambiguous. Unfortunately, Ada.Containers.Vectors has a number of such overloadings for Insert, Append, Prepend, and "&", so the problem may appear for any element type of a Vector that allows aggregates. For instance, if My_Vector is an instance of Ada.Containers.Vectors with an element type of Not_Lim as defined above, and V is an object of My_Vector.Vector, then My_Vector.Append (V, (Comp => 123)); will be illegal in Ada 2022 and legal in Ada 2012. This can easily be fixed by qualifying the aggregate with the correct type. 6.l/5

4.3.1 Record Aggregates

[In a record_aggregate, a value is specified for each component of the record or record extension value, using either a named or a positional association.] 1

Syntax

record_aggregate ::= (record_component_association_list) 2

record_component_association_list ::= 3
  record_component_association {, record_component_association}
 | **null record**

4/2 {*AI95-00287-01*} record_component_association ::=
[component_choice_list =>] expression
| component_choice_list => <>

5/5 {*AI12-0212-1*} component_choice_list ::=
*component*_selector_name {'|' *component*_selector_name}
| **others**

6 A record_component_association is a *named component association* if it has a component_choice_list; otherwise, it is a *positional component association*. Any positional component associations shall precede any named component associations. If there is a named association with a component_choice_list of **others**, it shall come last.

6.a **Discussion:** These rules were implied by the BNF in an early version of the RM9X, but it made the grammar harder to read, and was inconsistent with how we handle discriminant constraints. Note that for array aggregates we still express some of the rules in the grammar, but array aggregates are significantly different because an array aggregate is either all positional (with a possible **others** at the end), or all named.

7 In the record_component_association_list for a record_aggregate, if there is only one association, it shall be a named association.

7.a/3 **Reason:** {*AI05-0264-1*} Otherwise, the construct would be interpreted as a parenthesized expression. This is considered a syntax rule, since it is relevant to overload resolution. We choose not to express it with BNF so we can share the definition of record_component_association_list in both record_aggregate and extension_aggregate.

7.b **Ramification:** The record_component_association_list of an extension_aggregate does not have such a restriction.

Name Resolution Rules

8/2 {*AI95-00287-01*} The expected type for a record_aggregate shall be a single record type or record extension.

8.a **Ramification:** This rule is used to resolve whether an aggregate is an array_aggregate or a record_aggregate. The presence of a **with** is used to resolve between a record_aggregate and an extension_aggregate.

9/5 {*AI12-0127-1*} For the record_component_association_list of a record_aggregate, all components of the composite value defined by the aggregate are *needed*[; for the association list of an extension_aggregate, only those components not determined by the ancestor expression or subtype are needed (see 4.3.2).] Each *component*_selector_name in a record_component_association of a record_aggregate or extension_aggregate shall denote a needed component [(including possibly a discriminant)]. Each *component*_selector_name in a record_component_association of a record_delta_aggregate (see 4.3.4) shall denote a nondiscriminant component of the type of the aggregate.

9.a **Ramification:** For the association list of a record_aggregate, “needed components” includes every component of the composite value, but does not include those in unchosen variants (see AI83-309). If there are variants, then the value specified for the discriminant that governs them determines which variant is chosen, and hence which components are needed.

9.b If an extension defines a new known_discriminant_part, then all of its discriminants are needed in the component association list of an extension aggregate for that type, even if the discriminants have the same names and types as discriminants of the type of the ancestor expression. This is necessary to ensure that the positions in the record_-component_association_list are well defined, and that discriminants that govern variant_parts can be given by static expressions.

9.c/5 **Reason:** We don't define “needed” for record_delta_aggregates so that there is no completeness requirement. But that means that we need to ensure that the rules using “needed” doesn't appear to apply to record_delta_aggregates, and we also need Legality Rules to prevent giving the same component twice and giving components from two different variants.

9.d/5 **Term entry: needed component** — component of a record type or record extension that is required to have its value specified within a given aggregate

The expected type for the expression of a record_component_association is the type of the *associated* component(s); the associated component(s) are as follows: 10

- For a positional association, the component [(including possibly a discriminant)] in the corresponding relative position (in the declarative region of the type), counting only the needed components; 11

  Ramification: This means that for an association list of an extension_aggregate, only noninherited components are counted to determine the position. 11.a

  {*AI05-0005-1*} For a derived type (including type extensions), the order of declaration is defined in 3.4, “Derived Types and Classes”. In particular, all discriminants come first, regardless of whether they are defined for the parent type or are newly added to the derived type. 11.b/3

- For a named association with one or more *component*_selector_names, the named component(s); 12
- For a named association with the reserved word **others**, all needed components that are not associated with some previous association. 13

Legality Rules

If the type of a record_aggregate is a record extension, then it shall be a descendant of a record type, through one or more record extensions (and no private extensions). 14

{*AI05-0016-1*} {*AI12-0127-1*} A record_component_association_list shall be **null record** only if the list occurs in a record_aggregate or extension_aggregate, and there are no components needed for that list. 15/5

Ramification: For example, "(**null record**)" is a record_aggregate for a null record type. Similarly, "(T'(A) **with null record**)" is an extension_aggregate for a type defined as a null record extension of T. 15.a

{*AI05-0016-1*} If no components are needed and **null record** is not used, the record_component_association must necessarily be **others** => <>, as that is the only record_component_association that does not require an associated component. 15.b/3

{*AI95-00287-01*} {*AI05-0199-1*} {*AI12-0046-1*} {*AI12-0127-1*} For a record_aggregate or extension_aggregate, each record_component_association other than an **others** choice with a <> shall have at least one associated component, and each needed component shall be associated with exactly one record_component_association. For a record_delta_aggregate, each *component*_selector_name of each component_choice_list shall denote a distinct nondiscriminant component of the type of the aggregate. 16/5

Ramification: {*AI95-00287-01*} These rules apply to an association with an **others** choice with an expression. An **others** choice with a <> can match zero components or several components with different types. 16.a/2

Reason: {*AI95-00287-01*} Without these rules, there would be no way to know what was the expected type for the expression of the association. Note that some of the rules do not apply to <> associations, as we do not need to resolve anything. We allow **others** => <> to match no components as this is similar to array aggregates. That means that (**others** => <>) always represents a default-initialized record or array value. 16.b/2

This paragraph was deleted.{*AI12-0046-1*} 16.c/4

Ramification: The rule that requires at least one associated component for each record_component_association implies that there can be no extra associations for components that don't exist in the composite value, or that are already determined by the ancestor expression or subtype of an extension_aggregate. 16.d

The second part of the first sentence ensures that no needed components are left out, nor specified twice. 16.e

{*AI12-0046-1*} {*AI12-0127-1*} If a record_component_association with an expression has two or more associated components, all of them shall be of the same type, or all of them shall be of anonymous access types whose subtypes statically match. In addition, Legality Rules are enforced separately for each associated component. 16.1/5

Discussion: {*AI12-0046-1*} AI83-00244 also requires that the expression shall be legal for each associated component. Ada 95 omitted this wording, as it was thought that all cases of difference had been eliminated. That probably was true, but Ada 2005 reintroduced cases where the types match but the legality differs. For example: 16.f/4

16.g/4
```
type Rec (D : access Integer) is record
         F : access Integer;
end record;
```

16.h/5
```
begin
   declare
      X : aliased Integer;
      R : Rec := (D | F => X'Access); -- Legal for D, illegal for F
```

16.i/4 There are additional ways for this to happen; because of cases like the above we require that the Legality Rules are checked individually for each associated component.

17/5 {*AI05-0220-1*} {*AI12-0086-1*} {*AI12-0127-1*} For a record_aggregate or extension_aggregate, if a variant_part *P* is nested within a variant *V* that is not selected by the discriminant value governing the variant_part enclosing *V*, then there is no restriction on the discriminant governing *P*. Otherwise, the value of the discriminant that governs *P* shall be given by a static expression, or by a nonstatic expression having a constrained static nominal subtype. In this latter case of a nonstatic expression, there shall be exactly one discrete_choice_list of *P* that covers each value that belongs to the nominal subtype and satisfies the predicates of the subtype, and there shall be at least one such value.

17.a **Ramification:** This expression might either be given within the aggregate itself, or in a constraint on the parent subtype in a derived_type_definition for some ancestor of the type of the aggregate.

17.a.1/5 {*AI12-0086-1*} This means that in the nonstatic value case, the (static) subtype cannot have a null range.

17.1/2 {*AI95-00287-01*} A record_component_association for a discriminant without a default_expression shall have an expression rather than <>.

17.b/2 **Reason:** A discriminant must always have a defined value, but <> means uninitialized for a discrete type unless the component has a default value.

17.2/5 {*AI12-0127-1*} A record_component_association of the record_component_association_list of a record_delta_aggregate shall not:

17.3/5 - {*AI12-0418-1*} use the box compound delimiter <> rather than an expression;

17.4/5 - have an expression of a limited type;

17.5/5 - omit the component_choice_list; or

17.c/5 **Ramification:** {*AI12-0127-1*} A record_delta_aggregate cannot use positional notation; all choices shall be named choices.

17.6/5 - have a component_choice_list that is an **others** choice.

17.7/5 {*AI12-0127-1*} For a record_delta_aggregate, no two *component*_selector_names shall denote components declared within different variants of the same variant_part.

Dynamic Semantics

18 The evaluation of a record_aggregate consists of the evaluation of the record_component_association_list.

19/5 {*AI12-0086-1*} {*AI12-0451-1*} For the evaluation of a record_component_association_list, any per-object constraints (see 3.8) for components specified in the association list are elaborated and any expressions are evaluated and converted to the subtype of the associated component. Any constraint elaborations and expression evaluations (and conversions) occur in an arbitrary order, except that the expression for a discriminant is evaluated (and converted) prior to the elaboration of any per-object constraint that depends on it, which in turn occurs prior to the evaluation and conversion of the expression for the component with the per-object constraint. If the value of a discriminant that governs a selected variant_part is given by a nonstatic expression, and the evaluation of that expression yields a value that does not belong to the nominal subtype of the expression, then Constraint_Error is raised.

19.a **Ramification:** The conversion in the first rule might raise Constraint_Error.

19.b **Discussion:** This check in the first rule presumably happened as part of the dependent compatibility check in Ada 83.

Ramification: {*AI12-0086-1*} The test on the value of the discriminant can only fail if the value is outside the base range of its type, does not satisfy the (static) predicates of the subtype (possible when the predicate is disabled), or is an invalid representation. This is a rule similar to that used for case statements. As with case statements, this is not a "check"; it cannot be Suppressed. 19.c/5

{*AI95-00287-01*} For a record_component_association with an expression, the expression defines the value for the associated component(s). For a record_component_association with <>, if the component_declaration has a default_expression, that default_expression defines the value for the associated component(s); otherwise, the associated component(s) are initialized by default as for a stand-alone object of the component subtype (see 3.3.1). 19.1/2

The expression of a record_component_association is evaluated (and converted) once for each associated component. 20

Ramification: {*AI05-0005-1*} We don't need similar language for <>, as we're considering the value of <> for each individual component. Each component has its own default expression or its own default initialization (they can be different for each component; the components even could have different types), and each one has to be evaluated. So there is no need to repeat that. 20.a/3

NOTE {*AI12-0447-1*} By the rules given above, for a record_aggregate with positional associations, expressions specifying discriminant values appear first and in the same order as their corresponding discriminant_specifications, since the known_discriminant_part occurs first in the declaration of the type. 21/5

Examples

Example of a record aggregate with positional associations: 22

```
(4, July, 1776)                                    -- see 3.8
```
23

Examples of record aggregates with named associations: 24

```
(Day => 4, Month => July, Year => 1776)
(Month => July, Day => 4, Year => 1776)
```
25

```
(Disk, Closed, Track => 5, Cylinder => 12)         -- see 3.8.1
(Unit => Disk, Status => Closed, Cylinder => 9, Track => 1)
```
26

{*AI95-00287-01*} *Examples of component associations with several choices:* 27/2

```
(Value => 0, Succ|Pred => new Cell'(0, null, null))  -- see 3.10.1
```
28

```
   -- The allocator is evaluated twice:
   -- Succ and Pred designate different cells
```
29

```
(Value => 0, Succ|Pred => <>)                      -- see 3.10.1
```
29.1/2

```
   -- Succ and Pred will be set to null
```
29.2/2

Examples of record aggregates for tagged types (see 3.9 and 3.9.1): 30

```
Expression'(null record)
Literal'(Value => 0.0)
Painted_Point'(0.0, Pi/2.0, Paint => Red)
```
31

Extensions to Ada 83

Null record aggregates may now be specified, via "(**null record**)". However, this syntax is more useful for null record extensions in extension aggregates. 31.a

Wording Changes from Ada 83

Various AIs have been incorporated (AI83-00189, AI83-00244, and AI83-00309). In particular, Ada 83 did not explicitly disallow extra values in a record aggregate. Now we do. 31.b

Extensions to Ada 95

{*AI95-00287-01*} <> can be used in place of an expression in a record_aggregate, default initializing the component. 31.c/2

Wording Changes from Ada 95

{*AI95-00287-01*} Limited record_aggregates are allowed (since all kinds of aggregates can now be limited, see 4.3). 31.d/2

Incompatibilities With Ada 2005

31.e/3 {*AI05-0220-1*} **Correction:** Corrected wording so that the rule for discriminants governing variant_parts was not effectively circular. The change makes a few aggregates where a nonstatic discriminant governs an empty variant_part illegal. However, most Ada implementations already enforce some version of the new rule and already reject these aggregates. So it is unlikely that any incompatibility will be noticed in practice.

Extensions to Ada 2005

31.f/3 {*AI05-0016-1*} **Correction:** Fixed the wording so that **others** => <> can be used in place of **null record**. This is needed to avoid a generic contract issue for generic bodies: we do not want to have to assume the worst to disallow **others** => <> if the record type *might* be a null record.

31.g/3 {*AI05-0199-1*} **Correction:** We now allow multiple components with anonymous access types to be specified with a single component association. This is to be consistent with the capabilities of a named access type.

Extensions to Ada 2012

31.h/5 {*AI12-0086-1*} The value of a discriminant that governs a variant_part in an aggregate can be nonstatic if the nominal subtype of the discriminant expression is static and all possible values select a single variant.

Wording Changes from Ada 2012

31.i/4 {*AI12-0046-1*} **Corrigendum:** We explicitly say that the Legality Rules have to be rechecked for each component individually. This *seems* obvious, but as the AARM note 4.3.1 (16.c) appeared to say that this was not necessary, and since we explicitly state this sort of thing for generic instances, it seemed better to be explicit.

31.j/5 {*AI12-0127-1*} Made wording changes for delta_aggregates.

4.3.2 Extension Aggregates

1 [An extension_aggregate specifies a value for a type that is a record extension by specifying a value or subtype for an ancestor of the type, followed by associations for any components not determined by the ancestor_part.]

Language Design Principles

1.a The model underlying this syntax is that a record extension can also be viewed as a regular record type with an ancestor "prefix". The record_component_association_list corresponds to exactly what would be needed if there were no ancestor/prefix type. The ancestor_part determines the value of the ancestor/prefix.

Syntax

2 extension_aggregate ::=
(ancestor_part **with** record_component_association_list)

3 ancestor_part ::= expression | subtype_mark

Name Resolution Rules

4/2 {*AI95-00287-01*} The expected type for an extension_aggregate shall be a single type that is a record extension. If the ancestor_part is an expression, it is expected to be of any tagged type.

4.a **Reason:** We could have made the expected type *T*'Class where *T* is the ultimate ancestor of the type of the aggregate, or we could have made it even more specific than that. However, if the overload resolution rules get too complicated, the implementation gets more difficult and it becomes harder to produce good error messages.

4.b/3 **Ramification:** {*AI05-0005-1*} This rule is additive with the rule given in 4.3. That means the 4.3 rule must be satisfied even though it is always syntactically possible to tell that something is an extension aggregate rather than another kind of aggregate. Specifically, that means that an extension aggregate is ambiguous if the context is overloaded on array and/or untagged record types, even though those are never legal contexts for an extension aggregate. Thus, this rule acts more like a Legality Rule than a Name Resolution Rule.

Legality Rules

{*AI95-00306-01*} {*AI05-0115-1*} If the ancestor_part is a subtype_mark, it shall denote a specific tagged subtype. If the ancestor_part is an expression, it shall not be dynamically tagged. The type of the extension_aggregate shall be a descendant of the type of the ancestor_part (the *ancestor* type), through one or more record extensions (and no private extensions). If the ancestor_part is a subtype_mark, the view of the ancestor type from which the type is descended (see 7.3.1) shall not have unknown discriminants. 5/3

Reason: {*AI95-00306-01*} The expression cannot be dynamically tagged to prevent implicit "truncation" of a dynamically-tagged value to the specific ancestor type. This is similar to the rules in 3.9.2. 5.a/2

{*AI05-0067-1*} {*AI05-0244-1*} {*AI12-0236-1*} {*AI12-0317-1*} If the type of the ancestor_part is limited and at least one component is needed in the record_component_association_list, then the ancestor_part shall not have an operative constituent expression (see 4.4) that is a call to a function with an unconstrained result subtype. 5.1/5

Reason: {*AI05-0067-1*} {*AI05-0244-1*} This restriction simplifies implementation, because it ensures that either the caller or the callee knows the size to allocate for the aggregate. Without this restriction, information from both caller and callee would have to be combined to determine the appropriate size. 5.b/3

{*AI05-0067-1*} The (F(...) with null record) case is exempt from this rule, because such extension aggregates are created internally for inherited functions returning null-extension types — we can't very well make those illegal. Moreover, we don't need the rule for null extensions, as the result can simply use the space returned by the function call. 5.c/3

{*AI12-0317-1*} The mention of “operative constituents” means that constructs like parenthesized expressions, qualified_expressions, and conditional_expressions are ignored when enforcing this rule. 5.d/5

Static Semantics

For the record_component_association_list of an extension_aggregate, the only components *needed* are those of the composite value defined by the aggregate that are not inherited from the type of the ancestor_part, plus any inherited discriminants if the ancestor_part is a subtype_mark that denotes an unconstrained subtype. 6

Dynamic Semantics

For the evaluation of an extension_aggregate, the record_component_association_list is evaluated. If the ancestor_part is an expression, it is also evaluated; if the ancestor_part is a subtype_mark, the components of the value of the aggregate not given by the record_component_association_list are initialized by default as for an object of the ancestor type. Any implicit initializations or evaluations are performed in an arbitrary order, except that the expression for a discriminant is evaluated prior to any other evaluation or initialization that depends on it. 7

{*AI05-0282-1*} If the type of the ancestor_part has discriminants and the ancestor_part is not a subtype_mark that denotes an unconstrained subtype, then a check is made that each discriminant determined by the ancestor_part has the value specified for a corresponding discriminant, if any, either in the record_component_association_list, or in the derived_type_definition for some ancestor of the type of the extension_aggregate. Constraint_Error is raised if this check fails. 8/3

Ramification: Corresponding and specified discriminants are defined in 3.7. The rules requiring static compatibility between new discriminants of a derived type and the parent discriminant(s) they constrain ensure that at most one check is required per discriminant of the ancestor expression. 8.a

{*AI05-0282-1*} The check needs to be made any time that the ancestor is constrained; the source of the discriminants or the constraints is irrelevant. 8.b/3

NOTE 1 {*AI12-0447-1*} By the rules given in 4.3.1, if all components of the value of the extension_aggregate are determined by the ancestor_part, then the record_component_association_list will be simply **null record**. 9/5

10 NOTE 2 If the ancestor_part is a subtype_mark, then its type can be abstract. If its type is controlled, then as the last step of evaluating the aggregate, the Initialize procedure of the ancestor type is called, unless the Initialize procedure is abstract (see 7.6).

Examples

11 *Examples of extension aggregates (for types defined in 3.9.1):*

12
```
Painted_Point'(Point with Red)
(Point'(P) with Paint => Black)
```

13/5
```
{AI12-0178-1} (Expression with Left =>  new Literal'(Value => 1.2),
                          Right => new Literal'(Value => 3.4))
Addition'(Binop with null record)
                    -- presuming Binop is of type Binary_Operation
```

Extensions to Ada 83

13.a The extension aggregate syntax is new.

Incompatibilities With Ada 95

13.b/2 {*AI95-00306-01*} **Amendment Correction:** Eliminated implicit "truncation" of a dynamically tagged value when it is used as an ancestor expression. If an aggregate includes such an expression, it is illegal in Ada 2005. Such aggregates are thought to be rare; the problem can be fixed with a type conversion to the appropriate specific type if it occurs.

Wording Changes from Ada 95

13.c/2 {*AI95-00287-01*} Limited extension_aggregates are allowed (since all kinds of aggregates can now be limited, see 4.3).

Inconsistencies With Ada 2005

13.d/3 {*AI05-0282-1*} **Correction:** An extension_aggregate with an ancestor_part whose discriminants are constrained and inherited might now raise Constraint_Error if the aggregate's type is constrained, while it was OK in Ada 2005. In almost all cases, this will make no difference as the constraint will be checked by the immediately following use of the aggregate, but it is possible to compare such an aggregate for equality; in this case, no exception would be raised by Ada 2005, while Ada 2012 will raise Constraint_Error. This should be very rare, and having the possibility means that the representation of the aggregate type has to be able to support unconstrained values of the type, even if the first subtype is constrained and no such objects can be created any other way.

Incompatibilities With Ada 2005

13.e/3 {*AI05-0067-1*} **Correction:** A limited unconstrained ancestor expression that is a function call is now illegal unless the extension part is null. Such aggregates were first introduced in Ada 2005 and are very complex to implement as they must be built-in-place with an unknown size; as such, it is unlikely that they are implemented correctly in existing compilers and thus not often used in existing code.

13.f/3 {*AI05-0115-1*} **Correction:** An ancestor_part that is a subtype with unknown discriminants is now explicitly illegal. Such a subtype should not be used to declare an object, and the ancestor_part acts like an object. The Ada 95 rules did not disallow such cases, so it is possible that code exists that uses such an ancestor, but this should be rare.

Wording Changes from Ada 2012

13.g/5 {*AI12-0317-1*} Rewrote a Legality Rule to use the new term "operative constituent" in order to reduce duplication of text about ignoring parenthesized expressions and similar constructs.

4.3.3 Array Aggregates

1 [In an array_aggregate, a value is specified for each component of an array, either positionally or by its index.] For a positional_array_aggregate, the components are given in increasing-index order, with a final **others**, if any, representing any remaining components. For a named_array_aggregate, the components are identified by the values covered by the discrete_choices.

Language Design Principles

The rules in this subclause are based on terms and rules for discrete_choice_lists defined in 3.8.1, "Variant Parts and Discrete Choices". For example, the requirements that **others** come last and stand alone are found there. 1.a/1

Syntax

{*AI12-0306-1*} array_aggregate ::= 2/5
positional_array_aggregate | null_array_aggregate | named_array_aggregate

{*AI95-00287-01*} {*AI12-0212-1*} {*AI12-0306-1*} positional_array_aggregate ::= 3/5
(expression, expression {, expression})
| (expression {, expression}, **others** => expression)
| (expression {, expression}, **others** => <>)
| '[' expression {, expression}[, **others** => expression] ']'
| '[' expression {, expression}, **others** => <> ']'

{*AI12-0306-1*} null_array_aggregate ::= '[' ']' 3.1/5

{*AI12-0127-1*} {*AI12-0212-1*} named_array_aggregate ::= 4/5
(array_component_association_list)
| '[' array_component_association_list ']'

{*AI12-0127-1*} array_component_association_list ::= 4.1/5
array_component_association {, array_component_association}

{*AI95-00287-01*} {*AI12-0061-1*} array_component_association ::= 5/5
discrete_choice_list => expression
| discrete_choice_list => <>
| iterated_component_association

{*AI12-0061-1*} {*AI12-0212-1*} iterated_component_association ::= 5.1/5
for defining_identifier **in** discrete_choice_list => expression
| **for** iterator_specification => expression

An *n-dimensional* array_aggregate is one that is written as n levels of nested array_aggregates (or at the bottom level, equivalent string_literals). For the multidimensional case (n >= 2) the array_aggregates (or equivalent string_literals) at the n–1 lower levels are called *subaggregate*s of the enclosing n-dimensional array_aggregate. The expressions of the bottom level subaggregates (or of the array_aggregate itself if one-dimensional) are called the *array component expressions* of the enclosing n-dimensional array_aggregate. 6

Ramification: Subaggregates do not have a type. They correspond to part of an array. For example, with a matrix, a subaggregate would correspond to a single row of the matrix. The definition of "n-dimensional" array_aggregate applies to subaggregates as well as aggregates that have a type. 6.a

To be honest: An ***others*** *choice* is the reserved word **others** as it appears in a positional_array_aggregate or as the discrete_choice of the discrete_choice_list in an array_component_association. 6.b

{*AI12-0061-1*} The defining_identifier of an iterated_component_association declares an *index parameter*, an object of the corresponding index type. 6.1/5

Name Resolution Rules

{*AI95-00287-01*} The expected type for an array_aggregate (that is not a subaggregate) shall be a single array type. The component type of this array type is the expected type for each array component expression of the array_aggregate. 7/2

7.a/2 **Ramification:** {*AI95-00287-01*} We already require a single array or record type or record extension for an aggregate. The above rule requiring a single array type (and similar ones for record and extension aggregates) resolves which kind of aggregate you have.

8 The expected type for each discrete_choice in any discrete_choice_list of a named_array_aggregate is the type of the *corresponding index*; the corresponding index for an array_aggregate that is not a subaggregate is the first index of its type; for an (n–m)-dimensional subaggregate within an array_aggregate of an n-dimensional type, the corresponding index is the index in position m+1.

Legality Rules

9/5 {*AI12-0212-1*} An array_aggregate of an n-dimensional array type shall be written as an n-dimensional array_aggregate, or as a null_array_aggregate.

9.a **Ramification:** In an m-dimensional array_aggregate [(including a subaggregate)], where m >= 2, each of the expressions has to be an (m–1)-dimensional subaggregate.

10/5 {*AI12-0418-1*} An **others** choice is allowed for an array_aggregate only if an *applicable index constraint* applies to the array_aggregate. [An applicable index constraint is a constraint provided by certain contexts that can be used to determine the bounds of the array value specified by an array_aggregate.] Each of the following contexts (and none other) defines an applicable index constraint:

11/4 - {*AI95-00318-02*} {*AI12-0157-1*} For an explicit_actual_parameter, an explicit_generic_actual_parameter, the expression of a return statement, the return expression of an expression function, the initialization expression in an object_declaration, or a default_-expression [(for a parameter or a component)], when the nominal subtype of the corresponding formal parameter, generic formal parameter, function return object, expression function return object, object, or component is a constrained array subtype, the applicable index constraint is the constraint of the subtype;

12 - For the expression of an assignment_statement where the name denotes an array variable, the applicable index constraint is the constraint of the array variable;

12.a **Reason:** This case is broken out because the constraint comes from the actual subtype of the variable (which is always constrained) rather than its nominal subtype (which might be unconstrained).

13 - For the operand of a qualified_expression whose subtype_mark denotes a constrained array subtype, the applicable index constraint is the constraint of the subtype;

14 - For a component expression in an aggregate, if the component's nominal subtype is a constrained array subtype, the applicable index constraint is the constraint of the subtype;

14.a **Discussion:** Here, the array_aggregate with **others** is being used within a larger aggregate.

14.1/5 - {*AI12-0127-1*} For the *base*_expression of a delta_aggregate, if the nominal subtype of the delta_aggregate is a constrained array subtype, the applicable index constraint is the constraint of the subtype;

15/3 - {*AI05-0147-1*} For a parenthesized expression, the applicable index constraint is that, if any, defined for the expression;

15.a **Discussion:** RM83 omitted this case, presumably as an oversight. We want to minimize situations where an expression becomes illegal if parenthesized.

15.1/5 - {*AI05-0147-1*} {*AI12-0236-1*} For a conditional_expression (see 4.5.7), the applicable index constraint for each *dependent*_expression is that, if any, defined for the conditional_expression;

15.2/5 - {*AI12-0236-1*} For a declare_expression (see 4.5.9), the applicable index constraint for the *body*_expression is that, if any, defined for the declare_expression.

The applicable index constraint *applies* to an array_aggregate that appears in such a context, as well as to any subaggregates thereof. In the case of an explicit_actual_parameter (or default_expression) for a call on a generic formal subprogram, no applicable index constraint is defined. 16

Reason: This avoids generic contract model problems, because only mode conformance is required when matching actual subprograms with generic formal subprograms. 16.a

{*AI05-0153-3*} {*AI12-0061-1*} {*AI12-0127-1*} The discrete_choice_list of an array_component_association (including an iterated_component_association) is allowed to have a discrete_choice that is a nonstatic choice_expression or that is a subtype_indication or range that defines a nonstatic or null range, only if it is the single discrete_choice of its discrete_choice_list, and either there is only one array_component_association in the enclosing array_component_association_list or the enclosing aggregate is an array_delta_aggregate[, not an array_aggregate]. 17/5

Discussion: We now allow a nonstatic **others** choice even if there are other array component expressions as well. 17.a

Ramification: {*AI12-0127-1*} We allow multiple dynamic choices in array_delta_aggregates, but only one dynamic choice per association even in that case. 17.b/5

{*AI12-0212-1*} Either all or none of the array_component_associations of an array_component_association_list shall be iterated_component_associations with an iterator_specification. 17.1/5

{*AI05-0262-1*} In a named_array_aggregate where all discrete_choices are static, no two discrete_choices are allowed to cover the same value (see 3.8.1); if there is no **others** choice, the discrete_choices taken together shall exactly cover a contiguous sequence of values of the corresponding index type. 18/3

Ramification: This implies that each component must be specified exactly once. See AI83-309. 18.a

Reason: {*AI05-0262-1*} This has to apply even if there is only one static discrete_choice; a single choice has to represent a contiguous range (a subtype_mark with a static predicate might represent a discontiguous set of values). If the (single) choice is a dynamic subtype, we don't need to make this check as no predicates are allowed (see 3.2.4) and thus the range has to be contiguous. 18.b/3

A bottom level subaggregate of a multidimensional array_aggregate of a given array type is allowed to be a string_literal only if the component type of the array type is a character type; each character of such a string_literal shall correspond to a defining_character_literal of the component type. 19

Static Semantics

A subaggregate that is a string_literal is equivalent to one that is a positional_array_aggregate of the same length, with each expression being the character_literal for the corresponding character of the string_literal. 20

To be honest: {*AI12-0306-1*} This is true even in cases where there is no corresponding legal positional_array_aggregate. 20.a/5

{*AI12-0061-1*} The subtype (and nominal subtype) of an index parameter is the corresponding index subtype. 20.1/5

Dynamic Semantics

{*AI12-0212-1*} For an array_aggregate that contains only array_component_associations that are iterated_component_associations with iterator_specifications, evaluation proceeds in two steps: 20.2/5

1. {*AI12-0212-1*} {*AI12-0250-1*} Each iterator_specification is elaborated (in an arbitrary order) and an iteration is performed solely to determine a maximum count for the number of values produced by the iteration; all of these counts are combined to determine the overall length of the array, and ultimately the limits on the bounds of the array (defined below); 20.3/5

20.4/5 2. {*AI12-0212-1*} {*AI12-0250-1*} {*AI12-0327-1*} A second iteration is performed for each of the iterator_specifications, in the order given in the aggregate, and for each value conditionally produced by the iteration (see 5.5 and 5.5.2), the associated expression is evaluated, its value is converted to the component subtype of the array type, and used to define the value of the next component of the array starting at the low bound and proceeding sequentially toward the high bound. A check is made that the second iteration results in an array length no greater than the maximum determined by the first iteration; Constraint_Error is raised if this check fails.

20.b/5 **To be honest:** {*AI12-0212-1*} {*AI12-0250-1*} Constraint_Error should be raised no later than when the iterations exceed the maximum array length; memory that doesn't belong to the aggregate temporary should not be overwritten.

21/5 {*AI12-0212-1*} The evaluation of any other array_aggregate of a given array type proceeds in two steps:

22 1. Any discrete_choices of this aggregate and of its subaggregates are evaluated in an arbitrary order, and converted to the corresponding index type;

23 2. The array component expressions of the aggregate are evaluated in an arbitrary order and their values are converted to the component subtype of the array type; an array component expression is evaluated once for each associated component.

23.a **Ramification:** Subaggregates are not separately evaluated. The conversion of the value of the component expressions to the component subtype might raise Constraint_Error.

23.b/3 {*AI05-0005-1*} We don't need to say that <> is evaluated once for each component, as <> means that each component is *initialized by default*. That means that the actions defined for default initialization are applied to each component individually. Initializing one component by default and copying that to the others would be an incorrect implementation in general (although it might be OK if the default initialization is known to be constant).

23.1/4 {*AI95-00287-01*} {*AI12-0084-1*} Each expression in an array_component_association defines the value for the associated component(s). For an array_component_association with <>, the associated component(s) are initialized to the Default_Component_Value of the array type if this aspect has been specified for the array type; otherwise, they are initialized by default as for a stand-alone object of the component subtype (see 3.3.1).

23.2/5 {*AI12-0061-1*} {*AI12-0212-1*} During an evaluation of the expression of an iterated_component_association with a discrete_choice_list, the value of the corresponding index parameter is that of the corresponding index of the corresponding array component. During an evaluation of the expression of an iterated_component_association with an iterator_specification, the value of the loop parameter of the iterator_specification is the value produced by the iteration (as described in 5.5.2).

23.c/5 **Ramification:** Taken together with the preceding rule that "The array component expressions of the aggregate are evaluated in an arbitrary order", this implies that an index parameter of an iterated_component_association can take on its values in an arbitrary order. This is different from other constructs, such as a loop_statement. In contrast, a loop parameter of an iterated_component_association takes its values in the order defined by the iteration, the same as a loop_statement.

24 The bounds of the index range of an array_aggregate [(including a subaggregate)] are determined as follows:

25 • For an array_aggregate with an **others** choice, the bounds are those of the corresponding index range from the applicable index constraint;

26 • For a positional_array_aggregate [(or equivalent string_literal)] without an **others** choice, the lower bound is that of the corresponding index range in the applicable index constraint, if defined, or that of the corresponding index subtype, if not; in either case, the upper bound is determined from the lower bound and the number of expressions [(or the length of the string_literal)];

26.1/5 • {*AI12-0212-1*} {*AI12-0306-1*} For a null_array_aggregate, bounds for each dimension are determined as for a positional_array_aggregate without an **others** choice that has no expressions for each dimension;

Reason: We need a separate rule to describe what happens for a multidimensional null_array_aggregate; we could've combined the single-dimension case with the positional_array_aggregate rule. 26.a/5

- {*AI12-0212-1*} {*AI12-0250-1*} For a named_array_aggregate containing only iterated_component_associations with an iterator_specification, the lower bound is determined as for a positional_array_aggregate without an **others** choice, and the upper bound is determined from the lower bound and the total number of values produced by the second set of iterations; 26.2/5

- {*AI12-0212-1*} For any other named_array_aggregate without an **others** choice, the bounds are determined by the smallest and largest index values covered by any discrete_choice_list. 27/5

Reason: We don't need to say that each index value has to be covered exactly once, since that is a ramification of the general rule on aggregates that each component's value has to be specified exactly once. 27.a

For an array_aggregate, a check is made that the index range defined by its bounds is compatible with the corresponding index subtype. 28

Discussion: In RM83, this was phrased more explicitly, but once we define "compatibility" between a range and a subtype, it seems to make sense to take advantage of that definition. 28.a

Ramification: The definition of compatibility handles the special case of a null range, which is always compatible with a subtype. See AI83-00313. 28.b

{*AI05-0037-1*} For an array_aggregate with an **others** choice, a check is made that no expression or <> is specified for an index value outside the bounds determined by the applicable index constraint. 29/3

Discussion: RM83 omitted this case, apparently through an oversight. AI83-00309 defines this as a dynamic check, even though other Ada 83 rules ensured that this check could be performed statically. We now allow an **others** choice to be dynamic, even if it is not the only choice, so this check now needs to be dynamic, in some cases. Also, within a generic unit, this would be a nonstatic check in some cases. 29.a

For a multidimensional array_aggregate, a check is made that all subaggregates that correspond to the same index have the same bounds. 30

Ramification: No array bounds “sliding” is performed on subaggregates. 30.a

Reason: If sliding were performed, it would not be obvious which subaggregate would determine the bounds of the corresponding index. 30.b

The exception Constraint_Error is raised if any of the above checks fail. 31

Implementation Permissions

{*AI12-0212-1*} {*AI12-0250-1*} When evaluating iterated_component_associations for an array_aggregate that contains only iterated_component_associations with iterator_specifications, the first step of evaluating an iterated_component_association can be omitted if the implementation can determine the maximum number of values by some other means. 32/5

Discussion: For instance, if the type of the aggregate is constrained, the implementation can (but does not have to) calculate the expected length from the constraint. 32.a/5

NOTE 1 {*AI05-0004-1*} {*AI12-0306-1*} {*AI12-0440-1*} In an array_aggregate delimited by parentheses, positional notation can only be used with two or more expressions; a single expression in parentheses is interpreted as a parenthesized expression. An array_aggregate delimited by square brackets can be used to specify an array with a single component. 33/5

NOTE 2 {*AI12-0061-1*} An index parameter is a constant object (see 3.3). 34/5

Examples

Examples of array aggregates with positional associations: 35

```
(7, 9, 5, 1, 3, 2, 4, 8, 6, 0)                                    36
Table'(5, 8, 4, 1, others => 0)  -- see 3.6
```

37 *Examples of array aggregates with named associations:*

38/5
```
{AI12-0306-1} (1 .. 5 => (1 .. 8 => 0.0))       -- two-dimensional
[1 .. N => new Cell]                -- N new cells, in particular for N = 0
```

39/5
```
{AI12-0306-1} Table'(2 | 4 | 10 => 1, others => 0)
Schedule'(Mon .. Fri => True,  others => False)  -- see 3.6
Schedule'[Wed | Sun  => False, others => True]
Vector'(1 => 2.5)                                -- single-component vector
```

40 *Examples of two-dimensional array aggregates:*

41
```
-- Three aggregates for the same value of subtype Matrix(1..2,1..3) (see 3.6):
```

42/5
```
{AI12-0306-1} ((1.1, 1.2, 1.3), (2.1, 2.2, 2.3))
(1 => [1.1, 1.2, 1.3], 2 => [2.1, 2.2, 2.3])
[1 => (1 => 1.1, 2 => 1.2, 3 => 1.3),
 2 => (1 => 2.1, 2 => 2.2, 3 => 2.3)]
```

43 *Examples of aggregates as initial values:*

44
```
A : Table := (7, 9, 5, 1, 3, 2, 4, 8, 6, 0);          -- A(1)=7, A(10)=0
B : Table := (2 | 4 | 10 => 1, others => 0);          -- B(1)=0, B(10)=1
C : constant Matrix := (1 .. 5 => (1 .. 8 => 0.0));
                                                  -- C'Last(1)=5, C'Last(2)=8
```

45
```
D : Bit_Vector(M .. N) := (M .. N => True);           -- see 3.6
E : Bit_Vector(M .. N) := (others => True);
F : String(1 .. 1) := (1 => 'F');  -- a one component aggregate: same as "F"
```

46/5
```
{AI12-0061-1} G : constant Matrix :=
    (for I in 1 .. 4 =>
       (for J in 1 .. 4 =>
          (if I=J then 1.0 else 0.0))); -- Identity matrix
```

47/5
```
{AI12-0312-1} Empty_Matrix : constant Matrix := []; -- A matrix without elements
```

48/2 *{AI95-00433-01} Example of an array aggregate with defaulted others choice and with an applicable index constraint provided by an enclosing record aggregate:*

49/5
```
{AI12-0178-1} Buffer'(Size => 50, Pos => 1, Value => ('x', others => <>)) -- see 3.7
```

Incompatibilities With Ada 83

49.a.1/1 In Ada 95, no applicable index constraint is defined for a parameter in a call to a generic formal subprogram; thus, some aggregates that are legal in Ada 83 are illegal in Ada 95. For example:

49.a.2/1
```
subtype S3 is String (1 .. 3);
...
generic
   with function F (The_S3 : in S3) return Integer;
package Gp is
   I : constant Integer := F ((1 => '!', others => '?'));
       -- The aggregate is legal in Ada 83, illegal in Ada 95.
end Gp;
```

49.a.3/1 This change eliminates generic contract model problems.

Extensions to Ada 83

49.a We now allow "named with others" aggregates in all contexts where there is an applicable index constraint, effectively eliminating what was RM83-4.3.2(6). Sliding never occurs on an aggregate with others, because its bounds come from the applicable index constraint, and therefore already match the bounds of the target.

49.b The legality of an **others** choice is no longer affected by the staticness of the applicable index constraint. This substantially simplifies several rules, while being slightly more flexible for the user. It obviates the rulings of AI83-00244 and AI83-00310, while taking advantage of the dynamic nature of the "extra values" check required by AI83-00309.

49.c Named array aggregates are permitted even if the index type is descended from a formal scalar type. See 4.9 and AI83-00190.

Wording Changes from Ada 83

We now separate named and positional array aggregate syntax, since, unlike other aggregates, named and positional associations cannot be mixed in array aggregates (except that an **others** choice is allowed in a positional array aggregate). 49.d

We have also reorganized the presentation to handle multidimensional and one-dimensional aggregates more uniformly, and to incorporate the rulings of AI83-00019, AI83-00309, etc. 49.e

Extensions to Ada 95

{*AI95-00287-01*} <> can be used in place of an expression in an array_aggregate, default-initializing the component. 49.f/2

Wording Changes from Ada 95

{*AI95-00287-01*} Limited array_aggregates are allowed (since all kinds of aggregates can now be limited, see 4.3). 49.g/2

{*AI95-00318-02*} Fixed aggregates to use the subtype of the return object of a function, rather than the result subtype, because they can be different for an extended_return_statement, and we want to use the subtype that's explicitly in the code at the point of the expression. 49.h/2

Inconsistencies With Ada 2005

{*AI05-0037-1*} **Correction:** Fixed so the check for components outside of the array applies to both expressions and <>s. As <> was a new feature in Ada 2005, there should be little existing code that depends on a <> component that is specified outside of the array (and that is nonsense anyway, that a compiler is likely to detect even without an explicit language rule disallowing it). 49.i/3

Wording Changes from Ada 2005

{*AI05-0147-1*} Added a definition of the applicable index constraint for conditional_expressions (which are new). 49.j/3

Inconsistencies With Ada 2012

{*AI05-0084-1*} **Corrigendum:** Fixed so that the Default_Component_Value (if any) is used to initialize components specified with <>. This is what users would expect, and all Ada 2012 implementation known at the time of this writing initialize with the Default_Component_Value, so it is unlikely that anyone will be affected by this inconsistency. 49.k/4

Extensions to Ada 2012

{*AI12-0061-1*} {*AI12-0212-1*} {*AI12-0250-1*} The iterated_component_association is new (more than 25 years after it was originally proposed). It has been extended to allow container iterators as well as the basic index iterators. 49.l/5

{*AI12-0212-1*} An array_aggregate can be surrounded by brackets rather than parentheses. This allows consistent writing of zero- and one-component positional array aggregates, and is consistent with the syntax for writing container aggregates. 49.m/5

{*AI12-0212-1*} {*AI12-0306-1*} A null_array_aggregate is new. 49.n/5

Wording Changes from Ada 2012

{*AI05-0157-1*} **Corrigendum:** Added expression functions to the contexts that provide an applicable index constraint, because expression functions are handled separately in static semantics and legality rules. 49.o/4

{*AI12-0127-1*} Made syntax and wording changes for delta_aggregates. 49.p/5

{*AI12-0236-1*} Added a definition of the applicable index constraint for declare_expressions (which are new). This allows **others** in array aggregates inside of declare_expressions. 49.q/5

4.3.4 Delta Aggregates

{*AI12-0127-1*} {*AI12-0324-1*} Evaluating a (record or array) delta aggregate yields a composite value that starts with a copy of another value of the same type and then assigns to some (but typically not all) components of the copy. 1/5

Syntax

{*AI12-0127-1*} delta_aggregate ::= record_delta_aggregate | array_delta_aggregate 2/5

3/5 {*AI12-0127-1*} record_delta_aggregate ::=
(*base*_expression **with delta** record_component_association_list)

4/5 {*AI12-0127-1*} {*AI12-0212-1*} array_delta_aggregate ::=
(*base*_expression **with delta** array_component_association_list)
| '[' *base*_expression **with delta** array_component_association_list ']'

Name Resolution Rules

5/5 {*AI12-0127-1*} The expected type for a record_delta_aggregate shall be a single descendant of a record type or record extension.

6/5 {*AI12-0127-1*} The expected type for an array_delta_aggregate shall be a single array type.

7/5 {*AI12-0127-1*} The expected type for the *base*_expression of any delta_aggregate is the type of the enclosing delta_aggregate.

8/5 {*AI12-0127-1*} [The Name Resolution Rules and Legality Rules for each record_component_association of a record_delta_aggregate are as defined in 4.3.1.]

9/5 {*AI12-0127-1*} For an array_delta_aggregate, the expected type for each discrete_choice in an array_component_association is the index type of the type of the delta_aggregate.

10/5 {*AI12-0127-1*} The expected type of the expression in an array_component_association is defined as for an array_component_association occurring within an array_aggregate of the type of the delta_aggregate.

Legality Rules

11/5 {*AI12-0127-1*} For an array_delta_aggregate, the array_component_association shall not use the box symbol <>, and the discrete_choice shall not be **others**.

12/5 {*AI12-0127-1*} For an array_delta_aggregate, the dimensionality of the type of the delta_aggregate shall be 1.

13/5 {*AI12-0127-1*} For an array_delta_aggregate, the *base*_expression and each expression in every array_component_association shall be of a nonlimited type.

13.a/5 **Ramification:** The *base*_expression of a record_delta_aggregate may be of a limited type (for example a record with limited components), as it is not restricted. A rule in 4.3.1 ensures that we do not assign to a limited component. We do not allow any part of an array_delta_aggregate to be of a limited type, even the *base*_expression, as this is a useless construct (you would not be able to update anything because the components necessarily are also limited except in pathological cases).

Dynamic Semantics

14/5 {*AI12-0127-1*} {*AI12-0324-1*} {*AI12-0381-1*} The evaluation of a delta_aggregate begins with the evaluation of the *base*_expression of the delta_aggregate; then that value is used to create and initialize the anonymous object of the aggregate. The bounds of the anonymous object of an array_delta_aggregate and the discriminants (if any) of the anonymous object of a record_delta_aggregate are those of the *base*_expression. If a record_delta_aggregate is of a specific tagged type, its tag is that of the specific type; if it is of a class-wide type, its tag is that of the *base*_expression.

14.a/5 **Ramification:** This is the same anonymous object as described in 7.6, “Assignment and Finalization”; in particular, it might be required to be built in place.

14.b/5 For a specific tagged type, any extension components that the actual object underlying the *base*_expression might have are not used to initialize the anonymous object, which is critical if the aggregate is required to be built-in-place.

To be honest: The anonymous object associated with the evaluation of a delta_aggregate begins its life as a variable, not a constant (3.3 notwithstanding). This must be the case because the object is initialized and then subsequently modified. After evaluation of the delta_aggregate is complete, the object is a constant object. This is similar to the way that an extended return statement can provide a variable view of an object that will eventually be a constant object after the function returns its result. 14.c/5

{*AI12-0127-1*} For a record_delta_aggregate, for each component associated with each record_component_association (in an unspecified order): 15/5

- if the associated component belongs to a variant, a check is made that the values of the discriminants are such that the anonymous object has this component. The exception Constraint_Error is raised if this check fails. 16/5
- the expression of the record_component_association is evaluated, converted to the nominal subtype of the associated component, and assigned to the component of the anonymous object. 17/5

{*AI12-0127-1*} For an array_delta_aggregate, for each discrete_choice of each array_component_association (in the order given in the enclosing discrete_choice_list and array_component_association_list, respectively) the discrete_choice is evaluated; for each represented index value (in ascending order, if the discrete_choice represents a range): 18/5

- the index value is converted to the index type of the array type. 19/5
- a check is made that the index value belongs to the index range of the anonymous object of the aggregate; Constraint_Error is raised if this check fails. 20/5
- the component expression is evaluated, converted to the array component subtype, and assigned to the component of the anonymous object identified by the index value. 21/5

Reason: Unlike other aggregates, an array_delta_aggregate is evaluated in the order that it is written. This is necessary to get deterministic behavior, as (unlike other aggregates, including record_delta_aggregates) there is no requirement that the specified components be distinct. As such, the order requirement ensures that every array_delta_aggregate has a well-defined result, even if the same component is specified multiple times. 21.a/5

Examples

{*AI12-0127-1*} {*AI12-0429-1*} *Examples of use of delta aggregates in a postcondition:* 22/5

```
procedure Twelfth (D : in out Date) -- see 3.8 for type Date                    23/5
   with Post => D = (D'Old with delta Day => 12);

procedure The_Answer (V : in out Vector;                                        24/5
                      A, B : in Integer) -- see 3.6 for type Vector
   with Post => V = (V'Old with delta A .. B => 42.0, V'First => 0.0);
```

{*AI12-0127-1*} {*AI12-0324-1*} {*AI12-0379-1*} {*AI12-0386-1*} {*AI12-0429-1*} *Examples where the base expression is nontrivial:* 25/5

```
New_Cell : Cell := (Min_Cell (Head) with delta Value => 42);                    26/5
   -- see 3.10.1 for Cell and Head; 6.1 for Min_Cell

A1 : Vector := ((0 => 1.0, 1 => 2.0, 2 => 3.0)                                  27/5
       with delta Integer(Random * 2.0) => 14.2);
   -- see 3.6 for declaration of type Vector
   -- see 6.1 for declaration of Random

Tomorrow := ((Yesterday with delta Day => 12)                                   28/5
                  with delta Month => April); -- see 3.8
```

{*AI12-0127-1*} {*AI12-0379-1*} {*AI12-0429-1*} *Example where the base expression is class-wide:* 29/5

```
function Translate (P : Point'Class; X, Y : Real) return Point'Class is         30/5
   (P with delta X => P.X + X,
               Y => P.Y + Y); -- see 3.9 for declaration of type Point
```

Extensions to Ada 2012

{*AI12-0127-1*} Delta aggregates are new. 30.a/5

4.3.5 Container Aggregates

1/5 {*AI12-0212-1*} In a container_aggregate, values are specified for elements of a container; for a positional_container_aggregate, the elements are given sequentially; for a named_container_aggregate, the elements are specified by a sequence of key/value pairs, or using an iterator. The Aggregate aspect of the type of the aggregate determines how the elements are combined to form the container.

1.a/5 **Term entry: container aggregate** — construct used to define a value of a type that represents a collection of elements, by explicitly specifying the elements in the collection

2/5 {*AI12-0212-1*} For a type other than an array type, the following type-related operational aspect may be specified:

3/5 Aggregate This aspect is an aggregate of the form:

3.a/5 **Aspect Description for Aggregate:** Mechanism to define user-defined aggregates.

4/5
```
(Empty => name[,
 Add_Named => procedure_name][,
 Add_Unnamed => procedure_name][,
 New_Indexed => function_name,
 Assign_Indexed => procedure_name])
```

4.a/5 **Ramification:** {*AI22-0020-1*} As this aspect is described with syntax, it has to be given exactly as specified here. That means it cannot be given as a positional aggregate, nor can the order of the elements be changed. For instance, New_Indexed always has to occur before Assign_Indexed, and Empty always has to be first.

5/5 {*AI12-0212-1*} The type for which this aspect is specified is known as the *container type* of the Aggregate aspect. A *procedure*_name shall be specified for at least one of Add_Named, Add_Unnamed, or Assign_Indexed. If Add_Named is specified, neither Add_Unnamed nor Assign_Indexed shall be specified. Either both or neither of New_Indexed and Assign_Indexed shall be specified.

Name Resolution Rules

6/5 {*AI12-0212-1*} The name specified for Empty for an Aggregate aspect shall denote a constant of the container type, or denote exactly one function with a result type of the container type that has no parameters, or that has one **in** parameter of a signed integer type.

6.a/5 **Reason:** In the function case, the parameter, if present may be used to specify an initial size for the container, in anticipation of adding elements to it. For a positional aggregate, or a named aggregate that doesn't use an iterator, it will be initialized with the number of elements. For a named aggregate that uses an iterator, the implementation is permitted to estimate the number of elements that the iterator will produce, but it is not required to do so.

7/5 {*AI12-0212-1*} {*AI12-0437-1*} The *procedure*_name specified for Add_Unnamed for an Aggregate aspect shall denote exactly one procedure that has two parameters, the first an **in out** parameter of the container type, and the second an **in** parameter of some nonlimited type, called the *element type* of the container type.

8/5 {*AI12-0212-1*} {*AI12-0437-1*} The *function*_name specified for New_Indexed for an Aggregate aspect shall denote exactly one function with a result type of the container type, and two parameters of the same discrete type, with that type being the *key type* of the container type.

8.a/5 **Reason:** The New_Indexed function is used instead of Empty as the first step of creating an aggregate that is initialized using the Assign_Indexed procedure.

9/5 {*AI12-0212-1*} {*AI12-0437-1*} The *procedure*_name specified for Add_Named or Assign_Indexed for an Aggregate aspect shall denote exactly one procedure that has three parameters, the first an **in out** parameter of the container type, the second an **in** parameter of a nonlimited type (the *key type* of the

container type), and the third, an **in** parameter of a nonlimited type that is called the *element type* of the container type.

Legality Rules

{*AI12-0212-1*} If the container type of an Aggregate aspect is a private type, the full type of the container type shall not be an array type. If the container type is limited, the name specified for Empty shall denote a function rather than a constant object. 10/5

{*AI12-0212-1*} For an Aggregate aspect, the key type of Assign_Indexed shall be the same type as that of the parameters of New_Indexed. Additionally, if both Add_Unnamed and Assign_Indexed are specified, the final parameters shall be of the same type — the element type of the container type. 11/5

Static Semantics

{*AI12-0212-1*} {*AI12-0388-1*} The Aggregate aspect is nonoverridable (see 13.1.1). 12/5

Syntax

{*AI12-0212-1*} container_aggregate ::= 13/5
  null_container_aggregate
 | positional_container_aggregate
 | named_container_aggregate

{*AI12-0212-1*} null_container_aggregate ::= '[' ']' 14/5

{*AI12-0212-1*} positional_container_aggregate ::= '[' expression{, expression} ']' 15/5

{*AI12-0212-1*} named_container_aggregate ::= '[' container_element_association_list ']' 16/5

Reason: {*AI12-0212-1*} Unlike other aggregates, container aggregates have to be surrounded with brackets rather than parentheses. Using brackets allows writing zero- and one-element positional aggregates. (Were we to use parentheses, a one-element positional aggregate would look the same as and would always be interpreted as a parenthesized expression.) Unlike the case for arrays, where it is always possible to give bounds to work around this gap, such an aggregate could not be written at all for a sequence type (such as a list) where there is no index or key to use. 16.a/5

{*AI12-0212-1*} container_element_association_list ::= 17/5
  container_element_association {, container_element_association}

{*AI12-0212-1*} container_element_association ::= 18/5
  key_choice_list => expression
 | key_choice_list => <>
 | iterated_element_association

{*AI12-0212-1*} key_choice_list ::= key_choice {'|' key_choice} 19/5

{*AI12-0212-1*} key_choice ::= *key*_expression | discrete_range 20/5

{*AI12-0212-1*} iterated_element_association ::= 21/5
  for loop_parameter_specification[**use** *key*_expression] => expression
 | **for** iterator_specification[**use** *key*_expression] => expression

Name Resolution Rules

{*AI12-0212-1*} {*AI12-0437-1*} The expected type for a container_aggregate shall be a single type for which the Aggregate aspect has been specified. The expected type for each expression of a container_aggregate is the element type of the expected type. 22/5

23/5 {*AI12-0212-1*} The expected type for a *key*_expression, or a discrete_range of a key_choice, is the key type of the expected type of the aggregate.

Legality Rules

24/5 {*AI12-0212-1*} The expected type for a positional_container_aggregate shall have an Aggregate aspect that includes a specification for an Add_Unnamed procedure or an Assign_Indexed procedure. The expected type for a named_container_aggregate that contains one or more iterated_element_associations with a *key*_expression shall have an Aggregate aspect that includes a specification for the Add_Named procedure. The expected type for a named_container_aggregate that contains one or more key_choice_lists shall have an Aggregate aspect that includes a specification for the Add_Named or Assign_Indexed procedure. [A null_container_aggregate can be of any type with an Aggregate aspect.]

25/5 {*AI12-0212-1*} A non-null container aggregate is called an *indexed aggregate* if the expected type *T* of the aggregate specifies an Assign_Indexed procedure in its Aggregate aspect, and either there is no Add_Unnamed procedure specified for the type, or the aggregate is a named_container_aggregate with a container_element_association that contains a key_choice_list or a loop_parameter_specification. The key type of an indexed aggregate is also called the *index type* of the aggregate.

26/5 {*AI12-0212-1*} A container_element_association with a <> rather than an expression, or with a key_choice that is a discrete_range, is permitted only in an indexed aggregate.

26.a/5 **Reason:** Only an element of an indexed aggregate can be left uninitialized, because the compiler can simply omit producing an Assign_Indexed operation for the given index. For other kinds of aggregates, there are no operations that add an element without giving it a value. We could have defined such (optional) operations, but they would have added significant complication to an already complex feature without adding much additional expressive power. Note that, to be consistent with array aggregates, we do not support specifying <> in an iterated_element_association.

27/5 {*AI12-0212-1*} For an iterated_element_association without a *key*_expression, if the aggregate is an indexed aggregate or the expected type of the aggregate specifies an Add_Named procedure in its Aggregate aspect, then the type of the loop parameter of the iterated_element_association shall be the same as the key type of the aggregate.

27.a/5 **Ramification:** If there is a *key*_expression in an iterated_element_association, it determines the key of each added key/value pair, rather than the loop parameter. But if there is no *key*_expression, the loop parameter itself is used as the key.

28/5 {*AI12-0212-1*} {*AI12-0250-1*} For a named_container_aggregate that is an indexed aggregate, all container_element_associations shall contain either a key_choice_list, or a loop_parameter_specification without a *key*_expression or iterator_filter. Furthermore, for such an aggregate, either:

29/5 - all key_choices shall be static expressions or static ranges, and every loop_parameter_specification shall have a discrete_subtype_definition that defines a non-null static range, and the set of values of the index type covered by the key_choices and the discrete_subtype_definitions shall form a contiguous range of values with no duplications; or

30/5 - there shall be exactly one container_element_association, and if it has a key_choice_list, the list shall have exactly one key_choice.

30.a/5 **Reason:** The above is trying to mimic the rules for named_array_aggregates, without **others**.

Dynamic Semantics

31/5 {*AI12-0212-1*} The evaluation of a container_aggregate starts by creating an anonymous object *A* of the expected type *T*, initialized as follows:

- if the aggregate is an indexed aggregate, from the result of a call on the New_Indexed function; the actual parameters in this call represent the lower and upper bound of the aggregate, and are determined as follows: 32/5
  - if the aggregate is a positional_container_aggregate, the lower bound is the low bound of the subtype of the key parameter of the Add_Indexed procedure, and the upper bound has a position number that is the sum of the position number of the lower bound and one less than the number of expressions in the aggregate; 33/5
  - if the aggregate is a named_container_aggregate, the lower bound is the lowest value covered by a key_choice_list or is the low bound of a range defined by a discrete_subtype_definition of a loop_parameter_specification; the upper bound is the highest value covered by a key_choice_list or is the high bound of a range defined by a discrete_subtype_definition of a loop_parameter_specification. 34/5
- if the aggregate is not an indexed aggregate, by assignment from the Empty constant, or from a call on the Empty function specified in the Aggregate aspect. In the case of an Empty function with a formal parameter, the actual parameter has the following value: 35/5
  - for a null_container_aggregate, the value zero; 36/5
  - for a positional_container_aggregate, the number of expressions; 37/5
  - for a named_container_aggregate without an iterated_element_association, the number of *key_*expressions; 38/5
  - for a named_container_aggregate where every iterated_element_association contains a loop_parameter_specification, the total number of elements specified by all of the container_element_associations; 39/5
  - otherwise, to an implementation-defined value. 40/5

    Implementation Note: This value ought to be an estimate for the number of elements in the aggregate, if one is available. If not, it is suggested to use the default value for the parameter if one exists, and zero otherwise. 40.a/5

    Implementation defined: The value of the parameter to Empty for some container_aggregates. 40.b/5

{*AI12-0212-1*} The evaluation then proceeds as follows: 41/5

- for a null_container_aggregate, the anonymous object *A* is the result; 42/5
- for a positional_container_aggregate of a type with a specified Add_Unnamed procedure, each expression is evaluated in an arbitrary order, and the Add_Unnamed procedure is invoked in sequence with the anonymous object *A* as the first parameter and the result of evaluating each expression as the second parameter, in the order of the expressions; 43/5
- for a positional_container_aggregate that is an indexed aggregate, each expression is evaluated in an arbitrary order, and the Assign_Indexed procedure is invoked in sequence with the anonymous object *A* as the first parameter, the key value as the second parameter, computed by starting with the low bound of the subtype of the key formal parameter of the Assign_Indexed procedure and taking the successor of this value for each successive expression, and the result of evaluating each expression as the third parameter; 44/5
- for a named_container_aggregate for a type with an Add_Named procedure in its Aggregate aspect, the container_element_associations are evaluated in an arbitrary order: 45/5
  - for a container_element_association with a key_choice_list, for each key_choice of the list in an arbitrary order, the key_choice is evaluated as is the expression of the container_element_association (in an arbitrary order), and the Add_Named procedure is invoked once for each value covered by the key_choice, with the anonymous object *A* as the first parameter, the value from the key_choice as the second parameter, and the result of evaluating the expression as the third parameter; 46/5

47/5 • {*AI12-0212-1*} {*AI12-0327-1*} for a container_element_association with an iterated_element_association, first the iterated_element_association is elaborated, then an iteration is performed, and for each value conditionally produced by the iteration (see 5.5 and 5.5.2) the Add_Named procedure is invoked with the anonymous object *A* as the first parameter, the result of evaluating the expression as the third parameter, and:

48/5 – if there is a *key*_expression, the result of evaluating the *key*_expression as the second parameter;

49/5 – otherwise, with the loop parameter as the second parameter;

50/5 • for a named_container_aggregate that is an indexed aggregate, the evaluation proceeds as above for the case of Add_Named, but with the Assign_Indexed procedure being invoked instead of Add_Named; in the case of a container_element_association with a <> rather than an expression, the corresponding call on Assign_Indexed is not performed, leaving the component as it was upon return from the New_Indexed function;

51/5 • for any other named_container_aggregate, the container_element_associations (which are necessarily iterated_element_associations) are evaluated in the order given; each such evaluation comprises two steps:

52/5 1. the iterated_element_association is elaborated;

53/5 2. {*AI12-0212-1*} {*AI12-0327-1*} an iteration is performed, and for each value conditionally produced by the iteration (see 5.5 and 5.5.2) the Add_Unnamed procedure is invoked, with the anonymous object *A* as the first parameter and the result of evaluating the expression as the second parameter.

53.a/5 **Ramification:** In this case, the value of the loop parameter is not directly relevant, though presumably it appears within the expression of the iterated_element_association.

Examples

54/5 {*AI12-0212-1*} {*AI12-0429-1*} *Examples of specifying the Aggregate aspect for a Set_Type, a Map_Type, and a Vector_Type:*

55/5
```
-- Set_Type is a set-like container type.
type Set_Type is private
   with Aggregate => (Empty       => Empty_Set,
                      Add_Unnamed => Include);
function Empty_Set return Set_Type;
```

56/5
```
subtype Small_Int is Integer range -1000..1000;
```

57/5
```
procedure Include (S : in out Set_Type; N : in Small_Int);
```

58/5
```
-- Map_Type is a map-like container type.
type Map_Type is private
   with Aggregate =>  (Empty     => Empty_Map,
                       Add_Named => Add_To_Map);
```

59/5
```
procedure Add_To_Map (M     : in out Map_Type;
                      Key   : in Integer;
                      Value : in String);
```

60/5
```
Empty_Map : constant Map_Type;
```

61/5
```
-- Vector_Type is an extensible array-like container type.
type Vector_Type is private
   with Aggregate => (Empty          => Empty_Vector,
                      Add_Unnamed    => Append_One,
                      New_Indexed    => New_Vector,
                      Assign_Indexed => Assign_Element);
```

62/5
```
function Empty_Vector (Capacity : Integer := 0) return Vector_Type;
```

63/5
```
procedure Append_One (V : in out Vector_Type; New_Item : in String);
```

```
   procedure Assign_Element (V     : in out Vector_Type;                    64/5
                             Index : in Positive;
                             Item  : in String);
   function New_Vector (First, Last : Positive) return Vector_Type         65/5
      with Pre => First = Positive'First;
      --  Vectors are always indexed starting at the
      --  lower bound of their index subtype.
-- Private part not shown.                                                 66/5
```

{*AI12-0212-1*} {*AI12-0429-1*} *Examples of container aggregates for Set_Type, Map_Type, and Vector_Type:* 67/5

```
-- Example aggregates using Set_Type.                                     68/5
S : Set_Type;

-- Assign the empty set to S:                                             69/5
S := [];

-- Is equivalent to:                                                      70/5
S := Empty_Set;

-- A positional set aggregate:                                            71/5
S := [1, 2];

-- Is equivalent to:                                                      72/5
S := Empty_Set;
Include (S, 1);
Include (S, 2);

-- A set aggregate with an iterated_element_association:                  73/5
S := [for Item in 1 .. 5 => Item * 2];

-- Is equivalent to:                                                      74/5
S := Empty_Set;
for Item in 1 .. 5 loop
   Include (S, Item * 2);
end loop;

-- A set aggregate consisting of two iterated_element_associations:       75/5
S := [for Item in 1 .. 5 => Item,
      for Item in 1 .. 5 => -Item];

{AI12-0379-1} --  Is equivalent (assuming set semantics) to:              76/5
S := Empty_Set;
for Item in 1 .. 5 loop
   Include (S, Item);
end loop;
for Item in -5 .. -1 loop
   Include (S, Item);
end loop;

-- Example aggregates using Map_Type.                                     77/5
M : Map_Type;

-- A simple named map aggregate:                                          78/5
M := [12 => "house", 14 => "beige"];

-- Is equivalent to:                                                      79/5
M := Empty_Map;
Add_To_Map (M, 12, "house");
Add_To_Map (M, 14, "beige");

-- Define a table of pairs:                                               80/5
type Pair is record
   Key : Integer;
   Value : access constant String;
end record;

Table : constant array(Positive range <>) of Pair :=                       81/5
   [(Key => 33, Value => new String'("a nice string")),
    (Key => 44, Value => new String'("an even better string"))];
```

82/5
```
-- A map aggregate using an iterated_element_association
-- and a key_expression, built from from a table of key/value pairs:
M := [for P of Table use P.Key => P.Value.all];
```

83/5
```
-- Is equivalent to:
M := Empty_Map;
for P of Table loop
   Add_To_Map (M, P.Key, P.Value.all);
end loop;
```

84/5
```
-- Create an image table for an array of integers:
Keys : constant array(Positive range <>) of Integer := [2, 3, 5, 7, 11];
```

85/5
```
-- A map aggregate where the values produced by the
-- iterated_element_association are of the same type as the key
-- (hence a separate key_expression is unnecessary):
M := [for Key of Keys => Integer'Image (Key)];
```

86/5
```
-- Is equivalent to:
M := Empty_Map;
for Key of Keys loop
   Add_To_Map (M, Key, Integer'Image (Key));
end loop;
```

87/5
```
-- Example aggregates using Vector_Type.
V : Vector_Type;
```

88/5
```
-- A positional vector aggregate:
V := ["abc", "def"];
```

89/5
```
-- Is equivalent to:
V := Empty_Vector (2);
Append_One (V, "abc");
Append_One (V, "def");
```

90/5
```
-- An indexed vector aggregate:
V := [1 => "this", 2 => "is", 3 => "a", 4 => "test"];
```

91/5
```
-- Is equivalent to:
V := New_Vector (1, 4);
Assign_Element (V, 1, "this");
Assign_Element (V, 2, "is");
Assign_Element (V, 3, "a");
Assign_Element (V, 4, "test");
```

Extensions to Ada 2012

91.a/5 {*AI12-0212-1*} Container aggregates are new.

4.4 Expressions

1/3 {*AI05-0147-1*} {*AI05-0158-1*} {*AI05-0176-1*} An *expression* is a formula that defines the computation or retrieval of a value. In this Reference Manual, the term "expression" refers to a construct of the syntactic category expression or of any of the following categories: choice_expression, choice_relation, relation, simple_expression, term, factor, primary, conditional_expression, quantified_expression.

Syntax

2 expression ::=
relation {**and** relation} | relation {**and then** relation}
| relation {**or** relation} | relation {**or else** relation}
| relation {**xor** relation}

2.1/3 {*AI05-0158-1*} choice_expression ::=
choice_relation {**and** choice_relation}
| choice_relation {**or** choice_relation}
| choice_relation {**xor** choice_relation}

| choice_relation {**and then** choice_relation}
| choice_relation {**or else** choice_relation}

{*AI05-0158-1*} choice_relation ::= 2.2/3
simple_expression [relational_operator simple_expression]

{*AI05-0158-1*} {*AI12-0022-1*} {*AI12-0039-1*} relation ::= 3/4
simple_expression [relational_operator simple_expression]
| *tested*_simple_expression [**not**] **in** membership_choice_list
| raise_expression

{*AI05-0158-1*} {*AI12-0212-1*} membership_choice_list ::= 3.1/5
membership_choice {'|' membership_choice}

{*AI05-0158-1*} {*AI12-0039-1*} membership_choice ::= 3.2/4
*choice*_simple_expression | range | subtype_mark

simple_expression ::= 4
[unary_adding_operator] term {binary_adding_operator term}

term ::= factor {multiplying_operator factor} 5

factor ::= primary [** primary] | **abs** primary | **not** primary 6

{*AI05-0003-1*} {*AI05-0147-1*} {*AI05-0176-1*} {*AI12-0236-1*} primary ::= 7/5
numeric_literal | **null** | string_literal | aggregate
| name | allocator | (expression)
| (conditional_expression) | (quantified_expression)
| (declare_expression)

Name Resolution Rules

A name used as a primary shall resolve to denote an object or a value. 8

Discussion: This replaces RM83-4.4(3). We don't need to mention named numbers explicitly, because the name of a named number denotes a value. We don't need to mention attributes explicitly, because attributes now denote (rather than yield) values in general. Also, the new wording allows attributes that denote objects, which should always have been allowed (in case the implementation chose to have such a thing). 8.a

Reason: It might seem odd that this is an overload resolution rule, but it is relevant during overload resolution. For example, it helps ensure that a primary that consists of only the identifier of a parameterless function is interpreted as a function_call rather than directly as a direct_name. 8.b

Static Semantics

Each expression has a type; it specifies the computation or retrieval of a value of that type. 9

{*AI12-0317-1*} A primary that is an expression surrounded by (and) is known as a *parenthesized expression*. 9.1/5

{*AI12-0317-1*} Every name or expression consists of one or more *operative constituent* names or expressions, only one of which is evaluated as part of evaluating the name or expression (the *evaluated operative constituent*). The operative constituents are determined as follows, according to the form of the expression (or name): 9.2/5

- if the expression is a conditional_expression, the operative constituents of its *dependent*_expressions; 9.3/5
- if the expression (or name) is a parenthesized expression, a qualified_expression, or a view conversion, the operative constituent(s) of its operand; 9.4/5
- if the expression is a declare_expression, the operative constituent(s) of its *body*_expression; 9.5/5

9.6/5 • otherwise, the expression (or name) itself.

9.7/5 {*AI12-0317-1*} In certain contexts, we specify that an operative constituent shall (or shall not) be *newly constructed*. This means the operative constituent shall (or shall not) be an aggregate or a function_call; in either case, a raise_expression is permitted.

9.a/5 **To be honest:** If an if_expression does not have an **else** clause, "True" is an operative constituent of the expression and it can be the evaluated operative constituent.

Dynamic Semantics

10 The value of a primary that is a name denoting an object is the value of the object.

10.1/5 {*AI12-0227-1*} An expression of a numeric universal type is evaluated as if it has type *root_integer* (for *universal_integer*) or *root_real* (otherwise) unless the context identifies a specific type (in which case that type is used).

10.a/5 **Ramification:** This has no effect for a static expression; its value may be arbitrarily small or large since no specific type is expected for any expression for which this rule specifies one of the root types. The only effect of this rule is to allow Constraint_Error to be raised if the value is outside of the base range of *root_integer* or *root_real* when the expression is not static.

10.b/5 **Reason:** This rule means that implementations don't have to support unlimited range math at run time for universal expressions. Note that universal expressions for which the context doesn't specify a specific type are quite rare; attribute prefixes and results are the only known cases. (For operators, 8.6 already specifies that the operator of a root type be used, which provides a specific type.)

Implementation Permissions

11 For the evaluation of a primary that is a name denoting an object of an unconstrained numeric subtype, if the value of the object is outside the base range of its type, the implementation may either raise Constraint_Error or return the value of the object.

11.a/5 **Ramification:** {*AI05-0299-1*} {*AI12-0449-1*} This means that if extra-range intermediates are used to hold the value of an object of an unconstrained numeric subtype, a Constraint_Error can be raised on a read of the object, rather than only on an assignment to it. Similarly, it means that computing the value of an object of such a subtype can be deferred until the first read of the object (presuming no side effects other than failing an Overflow_Check are possible). This permission is over and above that provided by 11.6, since this allows the Constraint_Error to move to a different handler.

11.b **Reason:** This permission is intended to allow extra-range registers to be used efficiently to hold parameters and local variables, even if they might need to be transferred into smaller registers for performing certain predefined operations.

11.c **Discussion:** There is no need to mention other kinds of primarys, since any Constraint_Error to be raised can be "charged" to the evaluation of the particular kind of primary.

Examples

12 *Examples of primaries:*

13
```
4.0                 -- real literal
Pi                  -- named number
(1 .. 10 => 0)      -- array aggregate
Sum                 -- variable
Integer'Last        -- attribute
Sine(X)             -- function call
Color'(Blue)        -- qualified expression
Real(M*N)           -- conversion
(Line_Count + 10)   -- parenthesized expression
```

Examples of expressions: 14

```
{AI95-00433-01} Volume                    -- primary                          15/2
not Destroyed                 -- factor
2*Line_Count                  -- term
-4.0                          -- simple expression
-4.0 + A                      -- simple expression
B**2 - 4.0*A*C                -- simple expression
R*Sin(θ)*Cos(φ)               -- simple expression
Password(1 .. 3) = "Bwv"      -- relation
Count in Small_Int            -- relation
Count not in Small_Int        -- relation
Index = 0 or Item_Hit         -- expression
(Cold and Sunny) or Warm      -- expression (parentheses are required)
A**(B**C)                     -- expression (parentheses are required)
```

Extensions to Ada 83

In Ada 83, **out** parameters and their nondiscriminant subcomponents are not allowed as primaries. These restrictions are eliminated in Ada 95. 15.a

In various contexts throughout the language where Ada 83 syntax rules had simple_expression, the corresponding Ada 95 syntax rule has expression instead. This reflects the inclusion of modular integer types, which makes the logical operators "**and**", "**or**", and "**xor**" more useful in expressions of an integer type. Requiring parentheses to use these operators in such contexts seemed unnecessary and potentially confusing. Note that the bounds of a range still have to be specified by simple_expressions, since otherwise expressions involving membership tests might be ambiguous. Essentially, the operation ".." is of higher precedence than the logical operators, and hence uses of logical operators still have to be parenthesized when used in a bound of a range. 15.b

Wording Changes from Ada 2005

{*AI05-0003-1*} Moved qualified_expression from primary to name (see 4.1). This allows the use of qualified_expressions in more places. 15.c/3

{*AI05-0147-1*} {*AI05-0176-1*} Added conditional_expression and quantified_expression to primary. 15.d/3

{*AI05-0158-1*} Expanded membership test syntax (see 4.5.2). 15.e/3

Inconsistencies With Ada 2012

{*AI12-0039-1*} **Corrigendum:** Revised membership syntax to eliminate ambiguities. In some cases, previously ambiguous membership expressions will now have an unambiguous meaning. If an Ada 2012 implementation chose the "wrong" meaning, the expression could silently change meaning. Virtually all such expressions will become illegal because of type mismatches (and thus be incompatible, not inconsistent). However, if the choices are all of a Boolean type, resolution might succeed. For instance, `A in B | C and D` now always means `(A in B | C) and D`, but the original Ada 2012 syntax would have allowed it to mean `A in B | (C and D)`. If a compiler allowed the expression and interpreted it as the latter, the meaning of the expression would silently change. We expect this to be extremely rare as membership operations on Boolean types are unlikely (and this can happen only in code written for Ada 2012). 15.f/4

Incompatibilities With Ada 2012

{*AI12-0039-1*} **Corrigendum:** The revised membership syntax will require parentheses in membership_choice_lists in some cases where the Ada 2012 grammar did not require them. For instance, `A in B in C | D` is now illegal. However, such expressions can be interpreted in multiple ways (either `A in (B in C) | D` or `A in (B in C | D)` for this example), so using such expressions is likely to be dangerous (another compiler might interpret the expression differently). In addition, all such expressions occur only in Ada 2012 syntax; so they should be rare. 15.g/4

Wording Changes from Ada 2012

{*AI12-0227-1*} **Correction:** Added wording so that universal expressions evaluated at run time can raise Constraint_Error if the value is outside of the range of *root_integer* or *root_real*. We don't document this as an inconsistency because the rule requires no implementation to change (as Constraint_Error is not required); it just allows implementations that already raise Constraint_Error (which is all of them surveyed) to be considered correct. 15.h/5

{*AI12-0236-1*} Added declare_expression to primary. It shares the rules about parentheses with conditional_expressions. 15.i/5

{*AI12-0317-1*} Added the definitions of “operative constituent” and “newly constructed” to centralize definitions that are needed for various rules and definitions across the Reference Manual. In particular, *operative constituent* is often 15.j/5

used when we want the semantics or legality to be unchanged by the presence of parens, qualification, or view conversions. Examples are found in 4.3.2, 6.2, and 7.5.

4.5 Operators and Expression Evaluation

1 [The language defines the following six categories of operators (given in order of increasing precedence). The corresponding operator_symbols, and only those, can be used as designators in declarations of functions for user-defined operators. See 6.6, “Overloading of Operators”.]

Syntax

2 logical_operator ::= **and** | **or** | **xor**

3 relational_operator ::= = | /= | < | <= | > | >=

4 binary_adding_operator ::= + | – | &

5 unary_adding_operator ::= + | –

6 multiplying_operator ::= * | / | **mod** | **rem**

7 highest_precedence_operator ::= ** | **abs** | **not**

7.a **Discussion:** Some of the above syntactic categories are not used in other syntax rules. They are just used for classification. The others are used for both classification and parsing.

Static Semantics

8 For a sequence of operators of the same precedence level, the operators are associated with their operands in textual order from left to right. Parentheses can be used to impose specific associations.

8.a **Discussion:** The left-associativity is not directly inherent in the grammar of 4.4, though in 1.1.4 the definition of the metasymbols {} implies left associativity. So this could be seen as redundant, depending on how literally one interprets the definition of the {} metasymbols.

8.b See the Implementation Permissions below regarding flexibility in reassociating operators of the same precedence.

9 For each form of type definition, certain of the above operators are *predefined*; that is, they are implicitly declared immediately after the type definition. For each such implicit operator declaration, the parameters are called Left and Right for *binary* operators; the single parameter is called Right for *unary* operators. [An expression of the form X op Y, where op is a binary operator, is equivalent to a function_call of the form "op"(X, Y). An expression of the form op Y, where op is a unary operator, is equivalent to a function_call of the form "op"(Y). The predefined operators and their effects are described in subclauses 4.5.1 through 4.5.6.]

Dynamic Semantics

10 [The predefined operations on integer types either yield the mathematically correct result or raise the exception Constraint_Error. For implementations that support the Numerics Annex, the predefined operations on real types yield results whose accuracy is defined in Annex G, or raise the exception Constraint_Error.]

10.a **To be honest:** Predefined operations on real types can “silently” give wrong results when the Machine_Overflows attribute is false, and the computation overflows.

10.b/5 **Proof:** For integer types, this is normatively stated in the Dynamic Semantics of 3.5.4. For floating point types, this is normatively stated at the end of the Implementation Requirements of G.2.1. For fixed point types, this is normatively stated at the end of the Implementation Requirements of G.2.3.

Implementation Requirements

The implementation of a predefined operator that delivers a result of an integer or fixed point type may raise Constraint_Error only if the result is outside the base range of the result type. 11

The implementation of a predefined operator that delivers a result of a floating point type may raise Constraint_Error only if the result is outside the safe range of the result type. 12

To be honest: An exception is made for exponentiation by a negative exponent in 4.5.6. 12.a

Implementation Permissions

For a sequence of predefined operators of the same precedence level (and in the absence of parentheses imposing a specific association), an implementation may impose any association of the operators with operands so long as the result produced is an allowed result for the left-to-right association, but ignoring the potential for failure of language-defined checks in either the left-to-right or chosen order of association. 13

Discussion: Note that the permission to reassociate the operands in any way subject to producing a result allowed for the left-to-right association is not much help for most floating point operators, since reassociation may introduce significantly different round-off errors, delivering a result that is outside the model interval for the left-to-right association. Similar problems arise for division with integer or fixed point operands. 13.a

Note that this permission does not apply to user-defined operators. 13.b

NOTE The two operands of an expression of the form X op Y, where op is a binary operator, are evaluated in an arbitrary order, as for any function_call (see 6.4). 14

Examples

Examples of precedence: 15

```
not Sunny or Warm      -- same as (not Sunny) or Warm                    16
X > 4.0 and Y > 0.0    -- same as (X > 4.0) and (Y > 0.0)

-4.0*A**2              -- same as –(4.0 * (A**2))                        17
abs(1 + A) + B         -- same as (abs (1 + A)) + B
Y**(-3)                -- parentheses are necessary
A / B * C              -- same as (A/B)*C
A + (B + C)            -- evaluate B + C before adding it to A
```

Wording Changes from Ada 83

We don't give a detailed definition of precedence, since it is all implicit in the syntax rules anyway. 17.a

The permission to reassociate is moved here from RM83-11.6(5), so it is closer to the rules defining operator association. 17.b

4.5.1 Logical Operators and Short-circuit Control Forms

Name Resolution Rules

An expression consisting of two relations connected by **and then** or **or else** (a *short-circuit control form*) shall resolve to be of some boolean type; the expected type for both relations is that same boolean type. 1

Reason: This rule is written this way so that overload resolution treats the two operands symmetrically; the resolution of overloading present in either one can benefit from the resolution of the other. Furthermore, the type expected by context can help. 1.a

Static Semantics

The following logical operators are predefined for every boolean type *T*, for every modular type *T*, and for every one-dimensional array type *T* whose component type is a boolean type: 2

```
function "and"(Left, Right : T) return T                                 3
function "or" (Left, Right : T) return T
function "xor"(Left, Right : T) return T
```

3.a/2 *This paragraph was deleted.*{*AI95-00145-01*}

3.b/2 **Ramification:** {*AI95-00145-01*} For these operators, we are talking about the type without any (interesting) subtype, and not some subtype with a constraint or exclusion. Since it's possible that there is no name for the "uninteresting" subtype, we denote the type with an italicized *T*. This applies to the italicized *T* in many other predefined operators and attributes as well.

3.c/2 {*AI95-00145-01*} In many cases, there is a subtype with the correct properties available. The italicized *T* means:

3.d/2 - *T*'Base, for scalars;

3.e/2 - the first subtype of *T*, for tagged types;

3.f/2 - a subtype of the type *T* without any constraint or null exclusion, in other cases.

3.g/2 Note that "without a constraint" is not the same as unconstrained. For instance, a record type with no discriminant part is considered constrained; no subtype of it has a constraint, but the subtype is still constrained.

3.h/2 Thus, the last case often is the same as the first subtype of *T*, but that isn't the case for constrained array types (where the correct subtype is unconstrained) and for access types with a null_exclusion (where the correct subtype does not exclude null).

3.i/2 This italicized *T* is used for defining operators and attributes of the language. The meaning is intended to be as described here.

4 For boolean types, the predefined logical operators **and**, **or**, and **xor** perform the conventional operations of conjunction, inclusive disjunction, and exclusive disjunction, respectively.

5 For modular types, the predefined logical operators are defined on a bit-by-bit basis, using the binary representation of the value of the operands to yield a binary representation for the result, where zero represents False and one represents True. If this result is outside the base range of the type, a final subtraction by the modulus is performed to bring the result into the base range of the type.

6 The logical operators on arrays are performed on a component-by-component basis on matching components (as for equality — see 4.5.2), using the predefined logical operator for the component type. The bounds of the resulting array are those of the left operand.

Dynamic Semantics

7 The short-circuit control forms **and then** and **or else** deliver the same result as the corresponding predefined **and** and **or** operators for boolean types, except that the left operand is always evaluated first, and the right operand is not evaluated if the value of the left operand determines the result.

8 For the logical operators on arrays, a check is made that for each component of the left operand there is a matching component of the right operand, and vice versa. Also, a check is made that each component of the result belongs to the component subtype. The exception Constraint_Error is raised if either of the above checks fails.

8.a **Discussion:** The check against the component subtype is per AI83-00535.

9 NOTE The conventional meaning of the logical operators is given by the following truth table:

10

A	B	(A **and** B)	(A **or** B)	(A **xor** B)
True	True	True	True	False
True	False	False	True	True
False	True	False	True	True
False	False	False	False	False

Examples

11 *Examples of logical operators:*

12
```
Sunny or Warm
Filter(1 .. 10) and Filter(15 .. 24)    -- see 3.6.1
```

Examples of short-circuit control forms: 13

```
Next_Car.Owner /= null and then Next_Car.Owner.Age > 25    -- see 3.10.1
N = 0 or else A(N) = Hit_Value
```
14

4.5.2 Relational Operators and Membership Tests

[The *equality operators* = (equals) and /= (not equals) are predefined for nonlimited types. The other relational_operators are the *ordering operators* < (less than), <= (less than or equal), > (greater than), and >= (greater than or equal). The ordering operators are predefined for scalar types, and for *discrete array types*, that is, one-dimensional array types whose components are of a discrete type. 1

Ramification: The equality operators are not defined for *every* nonlimited type — see below for the exact rule. 1.a

{*AI05-0262-1*} {*AI05-0269-1*} A *membership test*, using **in** or **not in**, determines whether or not a value belongs to any given subtype or range, is equal to any given value, has a tag that identifies a type that is covered by a given type, or is convertible to and has an accessibility level appropriate for a given access type. Membership tests are allowed for all types.] 2/3

Name Resolution Rules

{*AI95-00251-01*} {*AI05-0158-1*} The *tested type* of a membership test is determined by the membership_choices of the membership_choice_list. Either all membership_choices of the membership_choice_list shall resolve to the same type, which is the tested type; or each membership_choice shall be of an elementary type, and the tested type shall be covered by each of these elementary types. 3/3

{*AI05-0158-1*} {*AI12-0039-1*} {*AI12-0418-1*} If the tested type is tagged, then the *tested*_simple_expression shall resolve to be of a type that is convertible (see 4.6) to the tested type; if untagged, the expected type of the *tested*_simple_expression is the tested type. The expected type of a *choice*_simple_expression in a membership_choice, and of a simple_expression of a range in a membership_choice, is the tested type of the membership operation. 3.1/5

Reason: {*AI95-00230-01*} The part of the rule for untagged types is stated in a way that ensures that operands like a string literal are still legal as operands of a membership test. 3.a/2

{*AI95-00251-01*} {*AI12-0039-1*} The significance of "is convertible to" is that we allow the *tested*_simple_expression to be of any class-wide type that could be converted to the tested type, not just the one rooted at the tested type. This includes any class-wide type that covers the tested type, along with class-wide interfaces in some cases. 3.b/4

{*AI05-0158-1*} The special rule for determining the tested type for elementary types is to allow numeric literals in membership_choice_lists. Without the rule, `A in B | 1` would be illegal as B and 1 would have different types (the literal having type *universal integer*). 3.c/3

Legality Rules

{*AI12-0039-1*} For a membership test, if the *tested*_simple_expression is of a tagged class-wide type, then the tested type shall be (visibly) tagged. 4/4

Ramification: Untagged types covered by the tagged class-wide type are not permitted. Such types can exist if they are descendants of a private type whose full type is tagged. This rule is intended to avoid confusion since such derivatives don't have their "own" tag, and hence are indistinguishable from one another at run time once converted to a covering class-wide type. 4.a

{*AI05-0158-1*} {*AI12-0039-1*} {*AI12-0328-1*} If a membership test includes one or more *choice*_simple_expressions and the tested type of the membership test is limited, then the tested type of the membership test shall have a visible primitive equality operator; if the tested type of the membership test is nonlimited with a user-defined primitive equality operator that is defined at a point where the type is limited, the tested type shall be a record type or record extension. 4.1/5

4.b/3 **Reason:** {*AI05-0158-1*} A visible equality operator is required in order to avoid breaking privacy; that is, we don't want to depend on a hidden equality operator.

4.c/5 {*AI12-0328-1*} We make the membership test on the nonlimited view of a type illegal if it would use a different equality operator than what would be used for a limited view of the same type (and such a limited view is known to exist).

Static Semantics

5 The result type of a membership test is the predefined type Boolean.

6 The equality operators are predefined for every specific type *T* that is not limited, and not an anonymous access type, with the following specifications:

7
```
function "=" (Left, Right : T) return Boolean
function "/="(Left, Right : T) return Boolean
```

7.1/2 {*AI95-00230-01*} The following additional equality operators for the *universal_access* type are declared in package Standard for use with anonymous access types:

7.2/2
```
function "=" (Left, Right : universal_access) return Boolean
function "/="(Left, Right : universal_access) return Boolean
```

8 The ordering operators are predefined for every specific scalar type *T*, and for every discrete array type *T*, with the following specifications:

9
```
function "<" (Left, Right : T) return Boolean
function "<="(Left, Right : T) return Boolean
function ">" (Left, Right : T) return Boolean
function ">="(Left, Right : T) return Boolean
```

Name Resolution Rules

9.1/2 {*AI95-00230-01*} {*AI95-00420-01*} At least one of the operands of an equality operator for *universal_access* shall be of a specific anonymous access type. Unless the predefined equality operator is identified using an expanded name with prefix denoting the package Standard, neither operand shall be of an access-to-object type whose designated type is *D* or *D*'Class, where *D* has a user-defined primitive equality operator such that:

9.2/2 - its result type is Boolean;

9.3/3 - {*AI05-0020-1*} it is declared immediately within the same declaration list as *D* or any partial or incomplete view of *D*; and

9.4/2 - at least one of its operands is an access parameter with designated type *D*.

9.a/2 **Reason:** The first sentence prevents compatibility problems by ensuring that these operators are not used for named access types. Also, universal access types do not count for the purposes of this rule. Otherwise, equality expressions like (X = **null**) would be ambiguous for normal access types.

9.b/2 The rest of the rule makes it possible to call (including a dispatching call) user-defined "=" operators for anonymous access-to-object types (they'd be hidden otherwise), and to write user-defined "=" operations for anonymous access types (by making it possible to see the universal operator using the Standard prefix).

9.c/2 **Ramification:** We don't need a similar rule for anonymous access-to-subprogram types because they can't be primitive for any type. Note that any nonprimitive user-defined equality operators still are hidden by the universal operators; they'll have to be called with a package prefix, but they are likely to be very uncommon.

Legality Rules

9.5/2 {*AI95-00230-01*} At least one of the operands of the equality operators for *universal_access* shall be of type *universal_access*, or both shall be of access-to-object types, or both shall be of access-to-subprogram types. Further:

9.6/2 - When both are of access-to-object types, the designated types shall be the same or one shall cover the other, and if the designated types are elementary or array types, then the designated subtypes shall statically match;

- When both are of access-to-subprogram types, the designated profiles shall be subtype conformant. 9.7/2

Reason: We don't want to allow completely arbitrary comparisons, as we don't want to insist that all access types are represented in ways that are convertible to one another. For instance, a compiler could use completely separate address spaces or incompatible representations. Instead, we allow compares if there exists an access parameter to which both operands could be converted. Since the user could write such an subprogram, and any reasonable meaning for "=" would allow using it in such a subprogram, this doesn't impose any further restrictions on Ada implementations. 9.d/2

{*AI05-0123-1*} {*AI12-0101-1*} {*AI12-0352-1*} If the profile of an explicitly declared primitive equality operator of an untagged record type is type conformant with that of the corresponding predefined equality operator, the declaration shall occur before the type is frozen. In addition, no type shall have been derived from the untagged record type before the declaration of the primitive equality operator. In addition to the places where Legality Rules normally apply (see 12.3), this rule applies also in the private part of an instance of a generic unit. 9.8/5

Dynamic Semantics

For discrete types, the predefined relational operators are defined in terms of corresponding mathematical operations on the position numbers of the values of the operands. 10

For real types, the predefined relational operators are defined in terms of the corresponding mathematical operations on the values of the operands, subject to the accuracy of the type. 11

Ramification: For floating point types, the results of comparing *nearly* equal values depends on the accuracy of the implementation (see G.2.1, “Model of Floating Point Arithmetic” for implementations that support the Numerics Annex). 11.a

Implementation Note: On a machine with signed zeros, if the generated code generates both plus zero and minus zero, plus and minus zero must be equal by the predefined equality operators. 11.b

Two access-to-object values are equal if they designate the same object, or if both are equal to the null value of the access type. 12

Two access-to-subprogram values are equal if they are the result of the same evaluation of an Access attribute_reference, or if both are equal to the null value of the access type. Two access-to-subprogram values are unequal if they designate different subprograms. [It is unspecified whether two access values that designate the same subprogram but are the result of distinct evaluations of Access attribute_references are equal or unequal.] 13

Reason: This allows each Access attribute_reference for a subprogram to designate a distinct “wrapper” subprogram if necessary to support an indirect call. 13.a

{*AI05-0123-1*} For a type extension, predefined equality is defined in terms of the primitive [(possibly user-defined)] equals operator for the parent type and for any components that have a record type in the extension part, and predefined equality for any other components not inherited from the parent type. 14/3

Ramification: Two values of a type extension are not equal if there is a variant_part in the extension part and the two values have different variants present. This is a ramification of the requirement that a discriminant governing such a variant_part has to be a “new” discriminant, and so has to be equal in the two values for the values to be equal. Note that variant_parts in the parent part need not match if the primitive equals operator for the parent type considers them equal. 14.a

{*AI95-00349-01*} The full type extension's operation is used for a private extension. This follows as only full types have parent types; the type specified in a private extension is an ancestor, but not necessarily the parent type. For instance, in: 14.b/2

```
with Pak1;
package Pak2 is
   type Typ3 is new Pak1.Typ1 with private;
private
   type Typ3 is new Pak1.Typ2 with null record;
end Pak2;
```
14.c/2

14.d/2 the parent type is Pak1.Typ2, not Pak1.Typ1, and the equality operator of Pak1.Typ2 is used to create predefined equality for Typ3.

15/5 {*AI05-0123-1*} {*AI12-0328-1*} For a private type, if its full type is a record type or a record extension, predefined equality is defined in terms of the primitive equals operator of the full type; otherwise, predefined equality for the private type is that of its full type.

16 For other composite types, the predefined equality operators [(and certain other predefined operations on composite types — see 4.5.1 and 4.6)] are defined in terms of the corresponding operation on *matching components*, defined as follows:

17 • For two composite objects or values of the same non-array type, matching components are those that correspond to the same component_declaration or discriminant_specification;

18 • For two one-dimensional arrays of the same type, matching components are those (if any) whose index values match in the following sense: the lower bounds of the index ranges are defined to match, and the successors of matching indices are defined to match;

19 • For two multidimensional arrays of the same type, matching components are those whose index values match in successive index positions.

20 The analogous definitions apply if the types of the two objects or values are convertible, rather than being the same.

20.a **Discussion:** Ada 83 seems to omit this part of the definition, though it is used in array type conversions. See 4.6.

21 Given the above definition of matching components, the result of the predefined equals operator for composite types (other than for those composite types covered earlier) is defined as follows:

22 • If there are no components, the result is defined to be True;

23 • If there are unmatched components, the result is defined to be False;

24/3 • {*AI05-0123-1*} Otherwise, the result is defined in terms of the primitive equals operator for any matching components that are records, and the predefined equals for any other matching components.

24.a/3 **Reason:** {*AI05-0123-1*} This asymmetry between components with and without a record type is necessary to preserve most upward compatibility and corresponds with the corresponding situation with generics, where the predefined operations "reemerge" in a generic for non-record types, but do not for record types. Also, only tagged types support user-defined assignment (see 7.6), so only tagged types can fully handle levels of indirection in the implementation of the type. For untagged types, one reason for a user-defined equals operator might be to allow values with different bounds or discriminants to compare equal in certain cases. When such values are matching components, the bounds or discriminants will necessarily match anyway if the discriminants of the enclosing values match.

24.b **Ramification:** Two null arrays of the same type are always equal; two null records of the same type are always equal.

24.c/3 {*AI05-0123-1*} Note that if a composite object has a component of a floating point type, and the floating point type has both a plus and minus zero, which are considered equal by the predefined equality, then a block compare cannot be used for the predefined composite equality. Of course, with user-defined equals operators for components that are records, a block compare breaks down anyway, so this is not the only special case that requires component-by-component comparisons. On a one's complement machine, a similar situation might occur for integer types, since one's complement machines typically have both a plus and minus (integer) zero.

24.d/2 **To be honest:** {*AI95-00230-01*} For a component with an anonymous access type, "predefined equality" is that defined for the *universal_access* type (anonymous access types have no equality operators of their own).

24.e/3 {*AI05-0123-1*} For a component with a record type *T*, "the primitive equals operator" is the one with two parameters of *T* which returns Boolean. We're not talking about some random other primitive function named "=".

24.1/5 {*AI05-0123-1*} {*AI12-0413-1*} If the primitive equals operator for an untagged record type is abstract, then Program_Error is raised at the point of any call to that abstract subprogram[, implicitly as part of an equality operation on an enclosing composite object, or in an instance of a generic with a formal private type where the actual type is a record type with an abstract "="].

24.f/5 **Reason:** {*AI12-0413-1*} An explicit call to an abstract subprogram is generally illegal. This rule is needed in order to define the effect of a call in an instance of a generic body, or an implicit call such as a call that is part of the predefined

equality operation for an enclosing composite type that has a component of an untagged record type that has an abstract primitive equals operator. For tagged types, an abstract primitive equals operator is only allowed for an abstract type, and abstract types cannot be components, so this case does not occur.

{*8652/0016*} {*AI95-00123-01*} For any composite type, the order in which "=" is called for components is unspecified. Furthermore, if the result can be determined before calling "=" on some components, it is unspecified whether "=" is called on those components. 24.2/1

The predefined "/=" operator gives the complementary result to the predefined "=" operator. 25

Ramification: Furthermore, if the user defines an "=" operator that returns Boolean, then a "/=" operator is implicitly declared in terms of the user-defined "=" operator so as to give the complementary result. See 6.6. 25.a

{*AI05-0264-1*} For a discrete array type, the predefined ordering operators correspond to *lexicographic order* using the predefined order relation of the component type: A null array is lexicographically less than any array having at least one component. In the case of nonnull arrays, the left operand is lexicographically less than the right operand if the first component of the left operand is less than that of the right; otherwise, the left operand is lexicographically less than the right operand only if their first components are equal and the tail of the left operand is lexicographically less than that of the right (the *tail* consists of the remaining components beyond the first and can be null). 26/3

{*AI05-0269-1*} An *individual membership test* is the membership test of a single membership_choice. 26.1/3

{*AI05-0158-1*} {*AI12-0039-1*} For the evaluation of a membership test using **in** whose membership_choice_list has a single membership_choice, the *tested*_simple_expression and the membership_choice are evaluated in an arbitrary order; the result is the result of the individual membership test for the membership_choice. 27/4

{*AI05-0158-1*} {*AI12-0039-1*} For the evaluation of a membership test using **in** whose membership_choice_list has more than one membership_choice, the *tested*_simple_expression of the membership test is evaluated first and the result of the operation is equivalent to that of a sequence consisting of an individual membership test on each membership_choice combined with the short-circuit control form **or else**. 27.1/4

Ramification: {*AI05-0158-1*} This equivalence includes the evaluation of the membership_choices; evaluation stops as soon as an individual choice evaluates to True. 27.a.1/3

{*AI05-0158-1*} {*AI05-0269-1*} An individual membership test yields the result True if: 28/3

- {*AI05-0158-1*} {*AI05-0264-1*} {*AI12-0039-1*} {*AI12-0328-1*} The membership_choice is a *choice*_simple_expression, and the *tested*_simple_expression is equal to the value of the membership_choice. If the tested type is a record type or a record extension, or is limited at the point where the membership test occurs, the test uses the primitive equality for the type; otherwise, the test uses predefined equality. 28.1/5

  Reason: We use the predefined equality operator if the membership test occurs where the type is nonlimited if the type is not a record type or record extension. However, to avoid confusion, cases where a membership test could use different equality operators based on the view are illegal. 28.a/5

- {*AI05-0153-3*} {*AI05-0158-1*} {*AI12-0039-1*} The membership_choice is a range and the value of the *tested*_simple_expression belongs to the given range. 28.2/4

- {*AI05-0153-3*} {*AI05-0158-1*} {*AI12-0039-1*} {*AI12-0071-1*} The membership_choice is a subtype_mark, the tested type is scalar, the value of the *tested*_simple_expression belongs to the range of the named subtype, and the value satisfies the predicates of the named subtype. 29/4

  Ramification: {*AI05-0153-3*} The scalar membership test only does a range check and a predicate check. It does not perform any other check, such as whether a value falls in a “hole” of a “holey” enumeration type. The Pos attribute function can be used for that purpose. 29.a/3

  Even though Standard.Float is an unconstrained subtype, the test “X in Float” will still return False (presuming the evaluation of X does not raise Constraint_Error) when X is outside Float'Range. 29.b

30/4 • {*AI95-00231-01*} {*AI05-0153-3*} {*AI05-0158-1*} {*AI12-0039-1*} {*AI12-0071-1*} The membership_choice is a subtype_mark, the tested type is not scalar, the value of the *tested*_simple_expression satisfies any constraints of the named subtype, the value satisfies the predicates of the named subtype, and:

30.1/4 • {*AI95-00231-01*} {*AI12-0039-1*} if the type of the *tested*_simple_expression is class-wide, the value has a tag that identifies a type covered by the tested type;

30.a/4 **Ramification:** {*AI12-0039-1*} Note that the tag is not checked if the *tested*_simple_expression is of a specific type.

30.2/4 • {*AI95-00231-01*} {*AI05-0149-1*} {*AI12-0039-1*} if the tested type is an access type and the named subtype excludes null, the value of the *tested*_simple_expression is not null;

30.3/4 • {*AI05-0149-1*} {*AI12-0039-1*} if the tested type is a general access-to-object type, the type of the *tested*_simple_expression is convertible to the tested type and its accessibility level is no deeper than that of the tested type; further, if the designated type of the tested type is tagged and the *tested*_simple_expression is nonnull, the tag of the object designated by the value of the *tested*_simple_expression is covered by the designated type of the tested type.

30.b/5 **To be honest:** {*AI12-0005-1*} In some of these cases, the bulleted checks need to pass before any predicate check is executed. Otherwise, a predicate check could be performed on an object of the wrong type. Consider:

30.c/5
```
type Root is tagged null record;
type Ext is new Root with record Data : Integer; end record;
function Is_Even (Param : Ext) return Boolean is
   (Param.Data mod 2 = 0);
subtype Even_Ext is Ext
  with Dynamic_Predicate => Is_Even (Even_Ext);
function F (X : Root'Class) return Boolean is
   (X in Even_Ext);
Flag : Boolean := F (Root'(null record));
```

30.d/5 If the predicate check is performed before the tag check or regardless of the result of that check, Is_Even would be called on an object that does not have a Data component. Similar cases can be constructed for general access types.

31/3 {*AI05-0264-1*} Otherwise, the test yields the result False.

32 A membership test using **not in** gives the complementary result to the corresponding membership test using **in**.

32.a/4 **To be honest:** {*AI05-0158-1*} {*AI12-0039-1*} *X* **not in** *A* | *B* | *C* is intended to be exactly equivalent to **not** (*X* **in** *A* | *B* | *C*), including the order of evaluation of the *tested*_simple_expression and membership_choices.

Implementation Requirements

32.1/1 {*8652/0016*} {*AI95-00123-01*} For all nonlimited types declared in language-defined packages, the "=" and "/=" operators of the type shall behave as if they were the predefined equality operators for the purposes of the equality of composite types and generic formal types.

32.a.1/3 **Ramification:** {*AI95-00123-01*} {*AI05-0123-1*} If any language-defined types are implemented with a user-defined "=" operator, then either the full type must be a record type, or the compiler must use “magic” to implement equality for this type. A normal user-defined "=" operator for a non-record type does *not* meet this requirement.

33/2 *This paragraph was deleted.*{*AI95-00230-01*}

34 NOTE If a composite type has components that depend on discriminants, two values of this type have matching components if and only if their discriminants are equal. Two nonnull arrays have matching components if and only if the length of each dimension is the same for both.

Examples

35 *Examples of expressions involving relational operators and membership tests:*

36
```
X /= Y
```

37/5
```
{AI12-0178-1} {AI12-0425-1} A_String = "A"                    -- True (see 3.3.1)
"" < A_String and A_String < "Aa"      -- True
A_String < "Bb" and A_String < "A  "   -- True
```

```
{AI05-0264-1} My_Car = null          -- True if My_Car has been set to null (see 3.10.1)
My_Car = Your_Car            -- True if we both share the same car
My_Car.all = Your_Car.all    -- True if the two cars are identical
```
38/3

```
{AI05-0158-1} N not in 1 .. 10              -- range membership test
Today in Mon .. Fri          -- range membership test
Today in Weekday             -- subtype membership test (see 3.5.1)
Card in Clubs | Spades       -- list membership test (see 3.5.1)
Archive in Disk_Unit         -- subtype membership test (see 3.8.1)
Tree.all in Addition'Class   -- class membership test (see 3.9.1)
```
39/3

Extensions to Ada 83

Membership tests can be used to test the tag of a class-wide value. 39.a

Predefined equality for a composite type is defined in terms of the primitive equals operator for tagged components or the parent part. 39.b

Wording Changes from Ada 83

The term “membership test” refers to the relation "X in S" rather to simply the reserved word **in** or **not in**. 39.c

We use the term “equality operator” to refer to both the = (equals) and /= (not equals) operators. Ada 83 referred to = as *the* equality operator, and /= as the inequality operator. The new wording is more consistent with the ISO 10646 name for "=" (equals sign) and provides a category similar to “ordering operator” to refer to both = and /=. 39.d

We have changed the term “catenate” to “concatenate”. 39.e

Extensions to Ada 95

{*AI95-00230-01*} {*AI95-00420-01*} The *universal_access* equality operators are new. They provide equality operations (most importantly, testing against **null**) for anonymous access types. 39.f/2

Wording Changes from Ada 95

{*8652/0016*} {*AI95-00123-01*} **Corrigendum:** Wording was added to clarify that the order of calls (and whether the calls are made at all) on "=" for components is unspecified. Also clarified that "=" must compose properly for language-defined types. 39.g/2

{*AI95-00251-01*} Memberships were adjusted to allow interfaces which don't cover the tested type, in order to be consistent with type conversions. 39.h/2

Inconsistencies With Ada 2005

{*AI05-0123-1*} User-defined untagged record equality is now defined to compose and be used in generics. Any code which assumes that the predefined equality reemerges in generics and in predefined equals for composite types could fail. However, it is much more likely that this change will fix bugs, as the behavior that would be expected (the user-defined "=" is used) will be true in more cases. 39.i/3

{*AI05-0123-1*} If a composite type contains a component of an untagged record type with an abstract equality operation, calling "=" on the composite type will raise Program_Error, while in the past a result will be returned using the predefined equality. This is quite possible in ASIS programs; it will detect a bug in such programs but of course the programs will need to be fixed before they will work. 39.j/3

Incompatibilities With Ada 2005

{*AI05-0123-1*} Late and hidden overriding of equality for untagged record types is now prohibited. This is necessary to make composition of equality predictable. It should always be possible to move the overriding to an earlier spot where it will be legal. 39.k/3

Extensions to Ada 2005

{*AI05-0149-1*} Membership tests for valid accessibility levels and tag coverage by the designated type for general access types are new. 39.l/3

{*AI05-0153-3*} Membership tests now include a predicate check. 39.m/3

{*AI05-0158-1*} Membership tests now allow multiple choices. 39.n/3

Wording Changes from Ada 2005

39.o/3 {*AI05-0020-1*} **Correction:** Wording was added to clarify that *universal_access* "=" does not apply if an appropriate operator is declared for a partial or incomplete view of the designated type. Otherwise, adding a partial or incomplete view could make some "=" operators ambiguous.

Inconsistencies With Ada 2012

39.p/4 {*AI12-0101-1*} **Corrigendum:** Removed the incompatible rule preventing the declaration of "=" in the private part of a package specification for an untagged record type that completes a private type. Any code that calls the predefined "=" on the private type will now execute the body for the redefined "=" instead for the predefined "=". Eliminating the rule eliminates an unnecessary incompatibility (especially for programs that never call the predefined "="). Moreover, (like the composition of untagged record "=" in Ada 2012) this is more likely to fix bugs than cause them (who defines an "=" with a presumably different result and does not want clients to use it?).

Incompatibilities With Ada 2012

39.q/5 {*AI12-0328-1*} A membership test is now illegal if all of the following are True:

39.r/5 • The membership test has one or more *choice*_simple_expressions;

39.s/5 • The membership test occurs in a place where the tested type is nonlimited;

39.t/5 • The tested type has a limited view with a primitive "=" operator;

39.u/5 • The full type of the tested type is not a record type or a record extension.

39.v/5 In such a case, the limited and nonlimited views would use different equality operators, which would be confusing and would cause various semantic difficulties. We believe such cases to be quite rare, especially as such membership tests are new to Ada 2012. The workaround is to replace such memberships with equality tests (assuming that the primitive "=" is intended; the predefined "=" is hidden in such cases and an extra type is needed to make it accessible).

Wording Changes from Ada 2012

39.w/4 {*AI12-0039-1*} **Corrigendum:** Reworded membership tests to use the syntax items *tested*_simple_expression and *choice*_simple_expression. This was necessary to eliminate wording ambiguities introduced when the grammar was corrected to eliminate syntax ambiguities. (Both of the above are now simple_expressions, so merely talking about a simple_expression is insufficient.)

39.x/4 {*AI12-0071-1*} **Corrigendum:** Updated wording of the membership tests to use the new term "satisfies the predicates" (see 3.2.4).

39.y/5 {*AI12-0413-1*} **Correction:** Clarified that a call to an abstract equality in an instance body raises Program_Error.

4.5.3 Binary Adding Operators

Static Semantics

1 The binary adding operators + (addition) and – (subtraction) are predefined for every specific numeric type *T* with their conventional meaning. They have the following specifications:

2
```
function "+"(Left, Right : T) return T
function "-"(Left, Right : T) return T
```

3 The concatenation operators & are predefined for every nonlimited, one-dimensional array type *T* with component type *C*. They have the following specifications:

4
```
function "&"(Left : T; Right : T) return T
function "&"(Left : T; Right : C) return T
function "&"(Left : C; Right : T) return T
function "&"(Left : C; Right : C) return T
```

Dynamic Semantics

5 For the evaluation of a concatenation with result type *T*, if both operands are of type *T*, the result of the concatenation is a one-dimensional array whose length is the sum of the lengths of its operands, and whose components comprise the components of the left operand followed by the components of the right operand. If the left operand is a null array, the result of the concatenation is the right operand. Otherwise, the lower bound of the result is determined as follows:

- If the ultimate ancestor of the array type was defined by a constrained_array_definition, then the lower bound of the result is that of the index subtype; 6

  Reason: This rule avoids Constraint_Error when using concatenation on an array type whose first subtype is constrained. 6.a

- If the ultimate ancestor of the array type was defined by an unconstrained_array_definition, then the lower bound of the result is that of the left operand. 7

[The upper bound is determined by the lower bound and the length.] A check is made that the upper bound of the result of the concatenation belongs to the range of the index subtype, unless the result is a null array. Constraint_Error is raised if this check fails. 8

If either operand is of the component type *C*, the result of the concatenation is given by the above rules, using in place of such an operand an array having this operand as its only component (converted to the component subtype) and having the lower bound of the index subtype of the array type as its lower bound. 9

Ramification: The conversion might raise Constraint_Error. The conversion provides “sliding” for the component in the case of an array-of-arrays, consistent with the normal Ada 95 rules that allow sliding during parameter passing. 9.a

The result of a concatenation is defined in terms of an assignment to an anonymous object, as for any function call (see 6.5). 10

Ramification: This implies that value adjustment is performed as appropriate — see 7.6. We don't bother saying this for other predefined operators, even though they are all function calls, because this is the only one where it matters. It is the only one that can return a value having controlled parts. 10.a

NOTE As for all predefined operators on modular types, the binary adding operators + and – on modular types include a final reduction modulo the modulus if the result is outside the base range of the type. 11

Implementation Note: A full "modulus" operation need not be performed after addition or subtraction of modular types. For binary moduli, a simple mask is sufficient. For nonbinary moduli, a check after addition to see if the value is greater than the high bound of the base range can be followed by a conditional subtraction of the modulus. Conversely, a check after subtraction to see if a "borrow" was performed can be followed by a conditional addition of the modulus. 11.a

Examples

Examples of expressions involving binary adding operators: 12

```
Z + 0.1       -- Z has to be of a real type                                  13

"A" & "BCD"   -- concatenation of two string literals                        14
'A' & "BCD"   -- concatenation of a character literal and a string literal
'A' & 'A'     -- concatenation of two character literals
```

Inconsistencies With Ada 83

The lower bound of the result of concatenation, for a type whose first subtype is constrained, is now that of the index subtype. This is inconsistent with Ada 83, but generally only for Ada 83 programs that raise Constraint_Error. For example, the concatenation operator in 14.a

```
X : array(1..10) of Integer;                                                 14.b
begin
X := X(6..10) & X(1..5);
```

would raise Constraint_Error in Ada 83 (because the bounds of the result of the concatenation would be 6..15, which is outside of 1..10), but would succeed and swap the halves of X (as expected) in Ada 95. 14.c

Extensions to Ada 83

Concatenation is now useful for array types whose first subtype is constrained. When the result type of a concatenation is such an array type, Constraint_Error is avoided by effectively first sliding the left operand (if nonnull) so that its lower bound is that of the index subtype. 14.d

4.5.4 Unary Adding Operators

Static Semantics

1 The unary adding operators + (identity) and – (negation) are predefined for every specific numeric type *T* with their conventional meaning. They have the following specifications:

2
```
function "+"(Right : T) return T
function "-"(Right : T) return T
```

3 NOTE For modular integer types, the unary adding operator –, when given a nonzero operand, returns the result of subtracting the value of the operand from the modulus; for a zero operand, the result is zero.

4.5.5 Multiplying Operators

Static Semantics

1 The multiplying operators ***** (multiplication), / (division), **mod** (modulus), and **rem** (remainder) are predefined for every specific integer type *T*:

2
```
function "*"  (Left, Right : T) return T
function "/"  (Left, Right : T) return T
function "mod"(Left, Right : T) return T
function "rem"(Left, Right : T) return T
```

3 Signed integer multiplication has its conventional meaning.

4 Signed integer division and remainder are defined by the relation:

5
```
A = (A/B)*B + (A rem B)
```

6 where (A **rem** B) has the sign of A and an absolute value less than the absolute value of B. Signed integer division satisfies the identity:

7
```
(-A)/B = -(A/B) = A/(-B)
```

8/3 {*AI05-0260-1*} The signed integer modulus operator is defined such that the result of A **mod** B is either zero, or has the sign of B and an absolute value less than the absolute value of B; in addition, for some signed integer value N, this result satisfies the relation:

9
```
A = B*N + (A mod B)
```

10 The multiplying operators on modular types are defined in terms of the corresponding signed integer operators[, followed by a reduction modulo the modulus if the result is outside the base range of the type] [(which is only possible for the "*" operator)].

10.a **Ramification:** The above identity satisfied by signed integer division is not satisfied by modular division because of the difference in effect of negation.

11 Multiplication and division operators are predefined for every specific floating point type *T*:

12
```
function "*"(Left, Right : T) return T
function "/"(Left, Right : T) return T
```

13 The following multiplication and division operators, with an operand of the predefined type Integer, are predefined for every specific fixed point type *T*:

14
```
function "*"(Left : T; Right : Integer) return T
function "*"(Left : Integer; Right : T) return T
function "/"(Left : T; Right : Integer) return T
```

15 [All of the above multiplying operators are usable with an operand of an appropriate universal numeric type.] The following additional multiplying operators for *root_real* are predefined[, and are usable when

both operands are of an appropriate universal or root numeric type, and the result is allowed to be of type *root_real*, as in a number_declaration]:

Ramification: These operators are analogous to the multiplying operators involving fixed or floating point types where *root_real* substitutes for the fixed or floating point type, and *root_integer* substitutes for Integer. Only values of the corresponding universal numeric types are implicitly convertible to these root numeric types, so these operators are really restricted to use with operands of a universal type, or the specified root numeric types. 15.a

```
function "*"(Left, Right : root_real) return root_real
function "/"(Left, Right : root_real) return root_real
```
16

```
function "*"(Left : root_real; Right : root_integer) return root_real
function "*"(Left : root_integer; Right : root_real) return root_real
function "/"(Left : root_real; Right : root_integer) return root_real
```
17

Multiplication and division between any two fixed point types are provided by the following two predefined operators: 18

Ramification: {*AI12-0005-1*} *Universal_fixed* is the universal type for the class of fixed point types, meaning that these operators take operands of any fixed point type (not necessarily the same) and return a result that is implicitly (or explicitly) convertible to any fixed point type. 18.a/3

```
function "*"(Left, Right : universal_fixed) return universal_fixed
function "/"(Left, Right : universal_fixed) return universal_fixed
```
19

Name Resolution Rules

{*AI95-00364-01*} {*AI95-00420-01*} The above two fixed-fixed multiplying operators shall not be used in a context where the expected type for the result is itself *universal_fixed* [— the context has to identify some other numeric type to which the result is to be converted, either explicitly or implicitly]. Unless the predefined universal operator is identified using an expanded name with prefix denoting the package Standard, an explicit conversion is required on the result when using the above fixed-fixed multiplication operator if either operand is of a type having a user-defined primitive multiplication operator such that: 19.1/2

- {*AI05-0020-1*} {*AI05-0209-1*} it is declared immediately within the same declaration list as the type or any partial or incomplete view thereof; and 19.2/3
- both of its formal parameters are of a fixed-point type. 19.3/2

{*AI95-00364-01*} {*AI95-00420-01*} A corresponding requirement applies to the universal fixed-fixed division operator. 19.4/2

Discussion: The *small* of *universal_fixed* is infinitesimal; no loss of precision is permitted. However, fixed-fixed division is impractical to implement when an exact result is required, and multiplication will sometimes result in unanticipated overflows in such circumstances, so we require an explicit conversion to be inserted in expressions like A * B * C if A, B, and C are each of some fixed point type. 19.a/2

On the other hand, X := A * B; is permitted by this rule, even if X, A, and B are all of different fixed point types, since the expected type for the result of the multiplication is the type of X, which is necessarily not *universal_fixed*. 19.b/2

{*AI95-00364-01*} {*AI95-00420-01*} We have made these into Name Resolution rules to ensure that user-defined primitive fixed-fixed operators are not made unusable due to the presence of these universal fixed-fixed operators. But we do allow these operators to be used if prefixed by package Standard, so that they can be used in the definitions of user-defined operators. 19.c/2

Paragraph 20 was deleted.

Dynamic Semantics

The multiplication and division operators for real types have their conventional meaning. [For floating point types, the accuracy of the result is determined by the precision of the result type. For decimal fixed point types, the result is truncated toward zero if the mathematical result is between two multiples of the *small* of the specific result type (possibly determined by context); for ordinary fixed point types, if the mathematical result is between two multiples of the *small*, it is unspecified which of the two is the result.] 21

22 The exception Constraint_Error is raised by integer division, **rem**, and **mod** if the right operand is zero. [Similarly, for a real type *T* with *T*'Machine_Overflows True, division by zero raises Constraint_Error.]

23 NOTE 1 For positive A and B, A/B is the quotient and A **rem** B is the remainder when A is divided by B. The following relations are satisfied by the rem operator:

24
```
  A  rem (-B) =   A rem B
(-A) rem   B  = -(A rem B)
```

25 NOTE 2 For any signed integer K, the following identity holds:

26
```
A mod B   =   (A + K*B) mod B
```

27 The relations between signed integer division, remainder, and modulus are illustrated by the following table:

28
A	B	A/B	A **rem** B	A **mod** B	A	B	A/B	A **rem** B	A **mod** B
10	5	2	0	0	-10	5	-2	0	0
11	5	2	1	1	-11	5	-2	-1	4
12	5	2	2	2	-12	5	-2	-2	3
13	5	2	3	3	-13	5	-2	-3	2
14	5	2	4	4	-14	5	-2	-4	1

30
A	B	A/B	A **rem** B	A **mod** B	A	B	A/B	A **rem** B	A **mod** B
10	-5	-2	0	0	-10	-5	2	0	0
11	-5	-2	1	-4	-11	-5	2	-1	-1
12	-5	-2	2	-3	-12	-5	2	-2	-2
13	-5	-2	3	-2	-13	-5	2	-3	-3
14	-5	-2	4	-1	-14	-5	2	-4	-4

Examples

31 *Examples of expressions involving multiplying operators:*

32
```
I : Integer := 1;
J : Integer := 2;
K : Integer := 3;
```

33
```
X : Real := 1.0;                    -- see 3.5.7
Y : Real := 2.0;
```

34
```
F : Fraction := 0.25;               -- see 3.5.9
G : Fraction := 0.5;
```

35
Expression	*Value*	*Result Type*
I*J	2	*same as I and J, that is, Integer*
K/J	1	*same as K and J, that is, Integer*
K **mod** J	1	*same as K and J, that is, Integer*
X/Y	0.5	*same as X and Y, that is, Real*
F/2	0.125	*same as F, that is, Fraction*
3*F	0.75	*same as F, that is, Fraction*
0.75*G	0.375	*universal_fixed, implicitly convertible to any fixed point type*
Fraction(F*G)	0.125	*Fraction, as stated by the conversion*
Real(J)*Y	4.0	*Real, the type of both operands after conversion of J*

Incompatibilities With Ada 83

35.a.1/2 {*AI95-00364-01*} {*AI95-00420-01*} The universal fixed-fixed multiplying operators are now directly available (see below). Any attempt to use user-defined fixed-fixed multiplying operators will be ambiguous with the universal ones. The only way to use the user-defined operators is to fully qualify them in a prefix call. This problem was not documented during the design of Ada 95, and has been mitigated by Ada 2005.

Extensions to Ada 83

35.a Explicit conversion of the result of multiplying or dividing two fixed point numbers is no longer required, provided the context uniquely determines some specific fixed point result type. This is to improve support for decimal fixed point, where requiring explicit conversion on every fixed-fixed multiply or divide was felt to be inappropriate.

35.b The type *universal_fixed* is covered by *universal_real*, so real literals and fixed point operands may be multiplied or divided directly, without any explicit conversions required.

Wording Changes from Ada 83

We have used the normal syntax for function definition rather than a tabular format. 35.c

Incompatibilities With Ada 95

{*AI95-00364-01*} We have changed the resolution rules for the universal fixed-fixed multiplying operators to remove the incompatibility with Ada 83 discussed above. The solution is to hide the universal operators in some circumstances. As a result, some legal Ada 95 programs will require the insertion of an explicit conversion around a fixed-fixed multiply operator. This change is likely to catch as many bugs as it causes, since it is unlikely that the user wanted to use predefined operators when they had defined user-defined versions. 35.d/2

Wording Changes from Ada 2005

{*AI05-0020-1*} {*AI05-0209-1*} **Correction:** Wording was added to clarify that *universal_fixed* "*" and "/" does not apply if an appropriate operator is declared for a partial (or incomplete) view of the designated type. Otherwise, adding a partial (or incomplete) view could make some "*" and "/" operators ambiguous. 35.e/3

{*AI05-0260-1*} **Correction:** The wording for the **mod** operator was corrected so that a result of 0 does not have to have "the sign of B" (which is impossible if B is negative). 35.f/3

4.5.6 Highest Precedence Operators

Static Semantics

The highest precedence unary operator **abs** (absolute value) is predefined for every specific numeric type *T*, with the following specification: 1

```
function "abs"(Right : T) return T
```
2

The highest precedence unary operator **not** (logical negation) is predefined for every boolean type *T*, every modular type *T*, and for every one-dimensional array type *T* whose components are of a boolean type, with the following specification: 3

```
function "not"(Right : T) return T
```
4

The result of the operator **not** for a modular type is defined as the difference between the high bound of the base range of the type and the value of the operand. [For a binary modulus, this corresponds to a bit-wise complement of the binary representation of the value of the operand.] 5

The operator **not** that applies to a one-dimensional array of boolean components yields a one-dimensional boolean array with the same bounds; each component of the result is obtained by logical negation of the corresponding component of the operand (that is, the component that has the same index value). A check is made that each component of the result belongs to the component subtype; the exception Constraint_Error is raised if this check fails. 6

Discussion: The check against the component subtype is per AI83-00535. 6.a

The highest precedence *exponentiation* operator ** is predefined for every specific integer type *T* with the following specification: 7

```
function "**"(Left : T; Right : Natural) return T
```
8

Exponentiation is also predefined for every specific floating point type as well as *root_real*, with the following specification (where *T* is *root_real* or the floating point type): 9

```
function "**"(Left : T; Right : Integer'Base) return T
```
10

{*AI05-0088-1*} The right operand of an exponentiation is the *exponent*. The value of X**N with the value of the exponent N positive is the same as the value of X*X*...X (with N–1 multiplications) except that the multiplications are associated in an arbitrary order. With N equal to zero, the result is one. With the value of N negative [(only defined for a floating point operand)], the result is the reciprocal of the result using the absolute value of N as the exponent. 11/3

11.a **Ramification:** The language does not specify the order of association of the multiplications inherent in an exponentiation. For a floating point type, the accuracy of the result might depend on the particular association order chosen.

Implementation Permissions

12 The implementation of exponentiation for the case of a negative exponent is allowed to raise Constraint_Error if the intermediate result of the repeated multiplications is outside the safe range of the type, even though the final result (after taking the reciprocal) would not be. (The best machine approximation to the final result in this case would generally be 0.0.)

13 NOTE As implied by the specification given above for exponentiation of an integer type, a check is made that the exponent is not negative. Constraint_Error is raised if this check fails.

Inconsistencies With Ada 83

13.a.1/1 {*8652/0100*} {*AI95-00018-01*} The definition of "**" allows arbitrary association of the multiplications which make up the result. Ada 83 required left-to-right associations (confirmed by AI83-00137). Thus it is possible that "**" would provide a slightly different (and more potentially accurate) answer in Ada 95 than in the same Ada 83 program.

Wording Changes from Ada 83

13.a We now show the specification for "**" for integer types with a parameter subtype of Natural rather than Integer for the exponent. This reflects the fact that Constraint_Error is raised if a negative value is provided for the exponent.

Wording Changes from Ada 2005

13.b/3 {*AI05-0088-1*} **Correction:** The equivalence definition for "**" was corrected so that it does not imply that the operands are evaluated multiple times.

4.5.7 Conditional Expressions

1/3 {*AI05-0147-1*} {*AI05-0188-1*} {*AI05-0262-1*} A conditional_expression selects for evaluation at most one of the enclosed *dependent*_expressions, depending on a decision among the alternatives. One kind of conditional_expression is the if_expression, which selects for evaluation a *dependent*_expression depending on the value of one or more corresponding conditions. The other kind of conditional_expression is the case_expression, which selects for evaluation one of a number of alternative *dependent*_expressions; the chosen alternative is determined by the value of a *selecting*_expression.

Language Design Principles

1.a/3 {*AI05-0188-1*} As previously noted, there are two kinds of conditional_expression, if_expressions and case_expressions. Whenever possible, we have written the rules in terms of conditional_expressions to avoid duplication.

1.b/3 {*AI05-0147-1*} The rules for conditional_expressions have been designed as much as possible to work similarly to a parenthesized expression. The intent is that as much as possible, wherever a parenthesized expression would be allowed, a conditional_expression would be allowed, and it should work the same way.

Syntax

2/3 {*AI05-0188-1*} conditional_expression ::= if_expression | case_expression

3/3 {*AI05-0147-1*} {*AI05-0188-1*} if_expression ::=
if condition **then** *dependent*_expression
{**elsif** condition **then** *dependent*_expression}
[**else** *dependent*_expression]

4/3 {*AI05-0147-1*} condition ::= *boolean*_expression

5/3 {*AI05-0188-1*} case_expression ::=

case *selecting_*expression **is**
case_expression_alternative {,
case_expression_alternative}

{*AI05-0188-1*} case_expression_alternative ::= 6/3
when discrete_choice_list =>
*dependent_*expression

{*AI05-0147-1*} Wherever the Syntax Rules allow an expression, a conditional_expression may be used in place of the expression, so long as it is immediately surrounded by parentheses. 7/3

Discussion: {*AI05-0147-1*} The syntactic category conditional_expression appears only as a primary that is parenthesized. The above rule allows it to additionally be used in other contexts where it would be directly surrounded by parentheses. 7.a/3

The grammar makes the following directly legal: 7.b/3

```
A := (if X then Y else Z); -- parentheses required
A := B + (if X then Y else Z) + C; -- parentheses required
```
7.c/3

The following procedure calls are syntactically legal; the first uses the above rule to eliminate the redundant parentheses found in the second: 7.d/3

```
P(if X then Y else Z);
P((if X then Y else Z)); -- redundant parentheses
```
7.e/3

```
P((if X then Y else Z), Some_Other_Param);
P(Some_Other_Param, (if X then Y else Z));
P(Formal => (if X then Y else Z));
```
7.f/3

whereas the following are illegal: 7.g/3

```
P(if X then Y else Z, Some_Other_Param);
P(Some_Other_Param, if X then Y else Z);
P(Formal => if X then Y else Z);
```
7.h/3

because in these latter cases, the conditional_expression is not immediately surrounded by parentheses (which means on both sides!). 7.i/3

The English-language rule applies in all places that could surround an expression with parentheses, including pragma arguments, type conversion and qualified expression operands, and array index expressions. 7.j/3

This English-language rule could have been implemented instead by adding a nonterminal expression_within_parentheses, which would consist of expressions and conditional_expressions. Then, that could be used in all of the syntax which could consist of parens directly around an expression. We did not do that because of the large amount of change required. A complete grammar is given in AI05-0147-1. 7.k/3

Implementation Note: {*AI05-0147-1*} Implementers are cautioned to consider error detection when implementing the syntax for conditional_expressions. An if_expression and an if_statement are very similar syntactically, (as are a case_expression and a case_statement) and simple mistakes can appear to change one into the other, potentially causing errors to be moved far away from their actual location. The absence of **end if** to terminate an if_expression (and **end case** for a case_expression) also may make error handling harder. 7.l/3

Name Resolution Rules

{*AI05-0147-1*} If a conditional_expression is expected to be of a type *T*, then each *dependent_*expression of the conditional_expression is expected to be of type *T*. Similarly, if a conditional_expression is expected to be of some class of types, then each *dependent_*expression of the conditional_expression is subject to the same expectation. If a conditional_expression shall resolve to be of a type *T*, then each *dependent_*expression shall resolve to be of type *T*. 8/3

{*AI05-0147-1*} The possible types of a conditional_expression are further determined as follows: 9/3

- If the conditional_expression is the operand of a type conversion, the type of the conditional_expression is the target type of the conversion; otherwise, 10/3

Reason: This rule distributes an enclosing type conversion to the *dependent_*expressions. This means that 10.a/3

```
T(if C then A else B)
```
10.b/3

has the same semantics as 10.c/3

```
(if C then T(A) else T(B))
```
10.d/3

11/3 • If all of the *dependent*_expressions are of the same type, the type of the conditional_expression is that type; otherwise,

12/3 • If a *dependent*_expression is of an elementary type, the type of the conditional_expression shall be covered by that type; otherwise,

12.a/3 **Reason:** This rule supports the use of numeric literals and universal expressions within a conditional_expression.

13/3 • If the conditional_expression is expected to be of type *T* or shall resolve to type *T*, then the conditional_expression is of type *T*.

13.a/3 **Ramification:** If the type of the conditional_expression cannot be determined by one of these rules, then Name Resolution has failed for that expression, even if the *dependent*_expressions would resolve individually.

14/3 {*AI05-0147-1*} A condition is expected to be of any boolean type.

15/3 {*AI05-0188-1*} The expected type for the *selecting*_expression and the discrete_choices are as for case statements (see 5.4).

Legality Rules

16/3 {*AI05-0147-1*} {*AI05-0188-1*} All of the *dependent*_expressions shall be convertible (see 4.6) to the type of the conditional_expression.

17/3 {*AI05-0147-1*} {*AI05-0188-1*} {*AI05-0269-1*} If the expected type of a conditional_expression is a specific tagged type, all of the *dependent*_expressions of the conditional_expression shall be dynamically tagged, or none shall be dynamically tagged. In this case, the conditional_expression is dynamically tagged if all of the *dependent*_expressions are dynamically tagged, is tag-indeterminate if all of the *dependent*_expressions are tag-indeterminate, and is statically tagged otherwise.

18/3 {*AI05-0147-1*} {*AI05-0262-1*} If there is no **else** *dependent*_expression, the if_expression shall be of a boolean type.

19/3 {*AI05-0188-1*} {*AI05-0269-1*} All Legality Rules that apply to the discrete_choices of a case_statement (see 5.4) also apply to the discrete_choices of a case_expression except within an instance of a generic unit.

19.a/3 **Reason:** The exemption for a case expression that occurs in an instance allows the following example:

19.b/3
```
generic
   with function Int_Func return Integer;
package G is
   X : Float := (case Int_Func is
                    when Integer'First .. -1 => -1.0,
                    when 0 => 0.0,
                    when Positive => 1.0);
end G;
```

19.c/3
```
function Nat_Func return Natural is (123);
```

19.d/3
```
package I is new G (Int_Func => Nat_Func); -- Legal
```

19.e/3 Note that the Legality Rules still apply in the generic unit itself; they are just not enforced in an instance of the unit.

Dynamic Semantics

20/3 {*AI05-0147-1*} {*AI05-0188-1*} For the evaluation of an if_expression, the condition specified after **if**, and any conditions specified after **elsif**, are evaluated in succession (treating a final **else** as **elsif** True **then**), until one evaluates to True or all conditions are evaluated and yield False. If a condition evaluates to True, the associated *dependent*_expression is evaluated, converted to the type of the if_expression, and the resulting value is the value of the if_expression. Otherwise (when there is no **else** clause), the value of the if_expression is True.

20.a/3 **Ramification: Else** is required unless the if_expression has a boolean type, so the last sentence can only apply to if_expressions with a boolean type.

{*AI05-0188-1*} For the evaluation of a case_expression, the *selecting*_expression is first evaluated. If the value of the *selecting*_expression is covered by the discrete_choice_list of some case_expression_alternative, then the *dependent*_expression of the case_expression_alternative is evaluated, converted to the type of the case_expression, and the resulting value is the value of the case_expression. Otherwise (the value is not covered by any discrete_choice_list, perhaps due to being outside the base range), Constraint_Error is raised. 21/3

Examples

{*AI12-0429-1*} *Example of use of an* if_expression*:* 22/5

```
{AI12-0312-1} Put_Line ("Casey is " &                                                    23/5
      (if Casey.Sex = M then "Male" else "Female")); -- see 3.10.1
```

{*AI12-0429-1*} *Example of use of a* case_expression*:* 24/5

```
{AI12-0312-1} function Card_Color (Card : Suit) return Color is -- see 3.5.1            25/5
  (case Card is
      when Clubs   | Spades   => Black,
      when Hearts  | Diamonds => Red);
```

Extensions to Ada 2005

{*AI05-0147-1*} If expressions and case expressions are new. 25.a/3

4.5.8 Quantified Expressions

{*AI12-0158-1*} Quantified expressions provide a way to write universally and existentially quantified predicates over containers and arrays. 0.1/4

Syntax

{*AI05-0176-1*} quantified_expression ::= 1/3

for quantifier loop_parameter_specification => predicate
| **for** quantifier iterator_specification => predicate

quantifier ::= **all** | **some** 2/3

predicate ::= *boolean*_expression 3/3

{*AI05-0176-1*} Wherever the Syntax Rules allow an expression, a quantified_expression may be used in place of the expression, so long as it is immediately surrounded by parentheses. 4/3

Discussion: The syntactic category quantified_expression appears only as a primary that is parenthesized. The above rule allows it to additionally be used in other contexts where it would be directly surrounded by parentheses. This is the same rule that is used for conditional_expressions; see 4.5.7 for a detailed discussion of the meaning and effects of this rule. 4.a/3

Name Resolution Rules

{*AI05-0176-1*} The expected type of a quantified_expression is any Boolean type. The predicate in a quantified_expression is expected to be of the same type. 5/3

Dynamic Semantics

{*AI05-0176-1*} {*AI12-0158-1*} {*AI12-0327-1*} For the evaluation of a quantified_expression, the loop_parameter_specification or iterator_specification is first elaborated. The evaluation of a quantified_expression then performs an iteration, and evaluates the predicate for each value conditionally produced by the iteration (see 5.5 and 5.5.2). 6/5

Ramification: {*AI12-0327-1*} The order of evaluation of the predicates is that in which the values are produced, as specified in 5.5 or 5.5.2. 6.a/5

7/3 {*AI05-0176-1*} The value of the quantified_expression is determined as follows:

8/4 - {*AI12-0158-1*} If the quantifier is **all**, the expression is False if the evaluation of any predicate yields False; evaluation of the quantified_expression stops at that point. Otherwise (every predicate has been evaluated and yielded True), the expression is True. Any exception raised by evaluation of the predicate is propagated.

8.a/3 **Ramification:** The expression is True if the domain contains no values.

9/4 - {*AI12-0158-1*} If the quantifier is **some**, the expression is True if the evaluation of any predicate yields True; evaluation of the quantified_expression stops at that point. Otherwise (every predicate has been evaluated and yielded False), the expression is False. Any exception raised by evaluation of the predicate is propagated.

9.a/3 **Ramification:** The expression is False if the domain contains no values.

Examples

10/5 {*AI05-0176-1*} {*AI12-0429-1*} *Example of a quantified expression as a postcondition for a sorting routine on an array A with an index subtype T:*

11/3
```
Post => (A'Length < 2 or else
   (for all I in A'First .. T'Pred(A'Last) => A (I) <= A (T'Succ (I))))
```

12/5 {*AI05-0176-1*} {*AI12-0429-1*} {*AI12-0430-1*} *Example of use of a quantified expression as an assertion that a positive number N is composite (as opposed to prime):*

13/5
```
{AI12-0312-1} pragma Assert (for some X in 2 .. N when X * X <= N => N mod X = 0);
   -- see iterator_filter in 5.5
```

Extensions to Ada 2005

13.a/3 {*AI05-0176-1*} Quantified expressions are new.

Wording Changes from Ada 2012

13.b/4 {*AI12-0158-1*} **Corrigendum:** Revised the wording to make it clear that the semantics is short-circuited, and what the result is when there are no values for the loop parameter.

4.5.9 Declare Expressions

1/5 {*AI12-0236-1*} Declare expressions provide a way to declare local constants and object renamings in an expression context.

Syntax

2/5 {*AI12-0236-1*} declare_expression ::=
declare {declare_item}
begin *body*_expression

3/5 {*AI12-0236-1*} declare_item ::= object_declaration | object_renaming_declaration

3.a/5 **Reason:** We allow (**declare begin** expression) with no declare_items, for uniformity with block statements, which also allow a pointless **declare**.

4/5 {*AI12-0236-1*} Wherever the Syntax Rules allow an expression, a declare_expression may be used in place of the expression, so long as it is immediately surrounded by parentheses.

4.a/5 **Discussion:** The syntactic category declare_expression appears only as a primary that is parenthesized. The above rule allows it to additionally be used in other contexts where it would be directly surrounded by parentheses. This is the same rule that is used for conditional_expressions; see 4.5.7 for a detailed discussion of the meaning and effects of this rule.

Legality Rules

{*AI12-0236-1*} A declare_item that is an object_declaration shall declare a constant of a nonlimited type. 5/5

Reason: We disallow limited objects to avoid the horror of task waiting in the middle of an expression. The solution used for controlled objects (wait until the entire expression is finished) isn't appropriate for tasks. This restriction also eliminates build-in-place complexities and the need to activate tasks found in a declare_item. 5.a/5

{*AI12-0236-1*} {*AI12-0317-1*} A declare_item that is an object_renaming_declaration (see 8.5.1) shall not rename an object of a limited type if any operative constituent of the *object_*name is a value conversion or is newly constructed (see 4.4). 6/5

Reason: We disallow renaming limited temporary objects (like the result of a function call) for the same reasons as for stand-alone objects. We do allow renaming existing limited objects because these don't cause any problems. Note that one cannot directly rename an aggregate, parenthesized expression, conditional_expression, or declare_expression (these are not names), but any of these can appear inside of a qualified_expression or type_conversion (which are names and thus can be renamed). 6.a/5

{*AI12-0236-1*} The following are not allowed within a declare_expression: a declaration containing the reserved word **aliased**; the attribute_designator Access or Unchecked_Access; or an anonymous access type. 7/5

Reason: We do not want to define accessibility rules for declare_items, as nested declare_expressions cause complexities or usage warts. We want to keep this feature simple to use, understand, and implement. Thus, we do not allow any of the features that would require accessibility rules. 7.a/5

Name Resolution Rules

{*AI12-0236-1*} If a declare_expression is expected to be of a type *T*, then the *body_*expression is expected to be of type *T*. Similarly, if a declare_expression is expected to be of some class of types, then the *body_*expression is subject to the same expectation. If a declare_expression shall resolve to be of a type *T*, then the *body_*expression shall resolve to be of type *T*. 8/5

{*AI12-0236-1*} The type of a declare_expression is the type of the *body_*expression. 9/5

Dynamic Semantics

{*AI12-0236-1*} For the evaluation of a declare_expression, the declare_items are elaborated in order, and then the *body_*expression is evaluated. The value of the declare_expression is that of the *body_*expression. 10/5

Examples

{*AI12-0236-1*} {*AI12-0429-1*} *Example of use of a declare expression as a replacement postcondition for Ada.Containers.Vectors."&" (see A.18.2):* 11/5

```
{AI12-0236-1} with Post =>                                                    12/5
   (declare
      Result renames Vectors."&"'Result;
      Length : constant Count_Type := Left.Length + Right.Length;
    begin
      Result.Length = Length and then
      not Tampering_With_Elements_Prohibited (Result) and then
      not Tampering_With_Cursors_Prohibited (Result) and then
      Result.Capacity >= Length)
```

Extensions to Ada 2012

{*AI12-0236-1*} Declare expressions are new. 12.a/5

4.5.10 Reduction Expressions

1/5 {*AI12-0242-1*} {*AI12-0262-1*} Reduction expressions provide a way to map or transform a collection of values into a new set of values, and then summarize the values produced by applying an operation to reduce the set to a single value result. A reduction expression is represented as an attribute_reference of the reduction attributes Reduce or Parallel_Reduce.

1.a/5 **Term entry: reduction expression** — expression that defines how to map or transform a collection of values into a new set of values, and then summarize the values by applying an operation to reduce the set to a single value

Syntax

2/5 {*AI12-0242-1*} {*AI12-0262-1*} reduction_attribute_reference ::=
value_sequence'reduction_attribute_designator
| prefix'reduction_attribute_designator

3/5 {*AI12-0262-1*} {*AI12-0355-2*} value_sequence ::=
'[' [**parallel**[(chunk_specification)] [aspect_specification]]
iterated_element_association ']'

4/5 {*AI12-0262-1*} reduction_attribute_designator ::=
*reduction*_identifier(reduction_specification)

5/5 {*AI12-0262-1*} {*AI12-0348-1*} reduction_specification ::= *reducer*_name, *initial_value*_expression

6/5 {*AI12-0250-1*} {*AI12-0262-1*} The iterated_element_association of a value_sequence shall not have a *key*_expression, nor shall it have a loop_parameter_specification that has the reserved word **reverse**.

6.a/5 **Reason:** The intent is that the syntax matches as closely as possible array or container aggregate notation. Syntax that matches a loop_parameter_specification with the **reverse** reserved word would not be permitted in an array aggregate, so we disallow that here.

7/5 {*AI12-0262-1*} The chunk_specification, if any, of a value_sequence shall be an *integer*_simple_expression.

Name Resolution Rules

8/5 {*AI12-0262-1*} The expected type for a reduction_attribute_reference shall be a single nonlimited type.

9/5 {*AI12-0262-1*} In the remainder of this subclause, we will refer to nonlimited subtypes *Value_Type* and *Accum_Type* of a reduction_attribute_reference. These subtypes and interpretations of the names and expressions of a reduction_attribute_reference are determined by the following rules:

9.a/5 **Discussion:** *Accum_Type* represents the result of the reduction (the *accumulator* type), and *Value_Type* represents the type of the input values to the reduction.

10/5 - {*AI12-0262-1*} *Accum_Type* is a subtype of the expected type of the reduction_attribute_reference.

11/5 - {*AI12-0262-1*} A *reducer subprogram* is subtype conformant with one of the following specifications:

12/5
```
function Reducer(Accumulator : Accum_Type;
                 Value : Value_Type) return Accum_Type;
```

13/5
```
procedure Reducer(Accumulator : in out Accum_Type;
                  Value : in Value_Type);
```

14/5 - {*AI12-0262-1*} The *reducer*_name of a reduction_specification denotes a reducer subprogram.

15/5 - {*AI12-0262-1*} The expected type of an *initial_value*_expression of a reduction_specification is that of subtype *Accum_Type*.

- {*AI12-0250-1*} {*AI12-0262-1*} The expected type of the expression of the iterated_element_association of a value_sequence is that of subtype *Value_Type*. 16/5

Legality Rules

{*AI12-0262-1*} {*AI12-0348-1*} If the reduction_attribute_reference has a value_sequence with the reserved word **parallel**, the subtypes *Accum_Type* and *Value_Type* shall statically match. 17/5

{*AI12-0262-1*} {*AI12-0348-1*} If the identifier of a reduction_attribute_designator is Parallel_Reduce, the subtypes *Accum_Type* and *Value_Type* shall statically match. 18/5

Reason: For a reduction_attribute_reference with a value_sequence that does not have the reserved word **parallel** or has a prefix and the identifier of the reduction_attribute_designator is Reduce, the subtypes *Accum_Type* and *Value_Type* can be different because only one logical thread of control is presumed so there is no need to combine multiple results. 18.a/5

Static Semantics

{*AI12-0262-1*} A reduction_attribute_reference denotes a value, with its nominal subtype being the subtype of the first parameter of the subprogram denoted by the *reducer_*name. 19/5

Dynamic Semantics

{*AI12-0250-1*} {*AI12-0262-1*} {*AI12-0327-1*} {*AI12-0355-2*} For the evaluation of a value_sequence, the iterated_element_association, the chunk_specification, and the aspect_specification, if any, are elaborated in an arbitrary order. Next an iteration is performed, and for each value conditionally produced by the iteration (see 5.5 and 5.5.2), the associated expression is evaluated with the loop parameter having this value, which produces a result that is converted to Value_Type and is used to define the next value in the sequence. 20/5

{*AI12-0262-1*} If the value_sequence does not have the reserved word **parallel**, it is produced as a single sequence of values by a single logical thread of control. If the reserved word **parallel** is present in the value_sequence, the enclosing reduction_attribute_reference is a parallel construct, and the sequence of values is generated by a parallel iteration (as defined in 5.5, 5.5.1, and 5.5.2), as a set of non-empty, non-overlapping contiguous chunks (*subsequences*) with one logical thread of control (see Clause 9) associated with each subsequence. If there is a chunk_specification, it determines the maximum number of chunks, as defined in 5.5; otherwise the maximum number of chunks is implementation defined. 21/5

Implementation defined: The maximum number of chunks for a parallel reduction expression without a chunk_specification. 21.a/5

{*AI12-0262-1*} For a value_sequence V, the following attribute is defined: 22/5

V'Reduce(Reducer, Initial_Value) 23/5

{*AI12-0262-1*} {*AI12-0348-1*} This attribute represents a *reduction expression*, and is in the form of a reduction_attribute_reference.

{*AI12-0262-1*} {*AI12-0348-1*} The evaluation of a use of this attribute begins by evaluating the parts of the reduction_attribute_designator (the *reducer_*name Reducer and the *initial_value_*expression Initial_Value), in an arbitrary order. It then initializes the *accumulator* of the reduction expression to the value of the *initial_value_*expression (the *initial value*). The value_sequence V is then evaluated. 24/5

{*AI12-0262-1*} {*AI12-0348-1*} If the value_sequence does not have the reserved word **parallel**, each value of the value_sequence is passed, in order, as the second (Value) parameter to a call on Reducer, with the first (Accumulator) parameter being the prior value of the accumulator, saving the result as the new value of the accumulator. The reduction expression yields the final value of the accumulator. 25/5

26/5 {*AI12-0262-1*} If the reserved word **parallel** is present in a value_sequence, then the (parallel) reduction expression is a parallel construct and the sequence has been partitioned into one or more subsequences (see above) each with its own separate logical thread of control.

27/5 {*AI12-0262-1*} {*AI12-0348-1*} Each logical thread of control creates a local accumulator for processing its subsequence. The accumulator for a subsequence is initialized to the first value of the subsequence, and calls on Reducer start with the second value of the subsequence (if any). The result for the subsequence is the final value of its local accumulator.

28/5 {*AI12-0262-1*} {*AI12-0348-1*} After all logical threads of control of a parallel reduction expression have completed, Reducer is called for each subsequence, in the original sequence order, passing the local accumulator for that subsequence as the second (Value) parameter, and the overall accumulator [(initialized above to the initial value)] as the first (Accumulator) parameter, with the result saved back in the overall accumulator. The parallel reduction expression yields the final value of the overall accumulator.

29/5 {*AI12-0262-1*} If the evaluation of the value_sequence yields an empty sequence of values, the reduction expression yields the initial value.

30/5 {*AI12-0262-1*} {*AI12-0348-1*} If an exception is propagated by one of the calls on Reducer, that exception is propagated from the reduction expression. If different exceptions are propagated in different logical threads of control, one is chosen arbitrarily to be propagated from the reduction expression as a whole.

30.a/5 **Implementation Note:** {*AI12-0262-1*} For a reduction_attribute_reference that has a value_sequence without the reserved word **parallel** or a prefix where the identifier of the reduction_attribute_designator is Reduce (see below), generally the compiler can still choose to execute the reduction in parallel, presuming doing so would not change the results. However sequential execution is necessary if the subtypes of the parameters of Reducer do not statically match, since there is no subprogram identified in the construct that could be used for combining the results in parallel.

30.b/5 **Discussion:** {*AI12-0262-1*} We say the calls to Reducer that combine the results of parallel execution are sequentially ordered in increasing order because certain reductions, such as vector concatenation, can be non-commutative (but still associative) operations. In order to return a deterministic result for parallel execution that is consistent with sequential execution, we need to specify an order for the iteration, and for the combination of results from the logical threads of control. It is also necessary that combining calls to Reducer are issued sequentially with respect to each other, which may require extra synchronization if the calls to Reducer are being executed by different logical threads of control.

31/5 {*AI12-0242-1*} For a prefix X of an array type[(after any implicit dereference)], or that denotes an iterable container object (see 5.5.1), the following attributes are defined:

32/5 X'Reduce(Reducer, Initial_Value)

{*AI12-0242-1*} {*AI12-0348-1*} X'Reduce is a reduction expression that yields a result equivalent to replacing the prefix of the attribute with the value_sequence:

32.1/5
```
[for Item of X => Item]
```

33/5 X'Parallel_Reduce(Reducer, Initial_Value)

{*AI12-0242-1*} {*AI12-0348-1*} X'Parallel_Reduce is a reduction expression that yields a result equivalent to replacing the attribute identifier with Reduce and the prefix of the attribute with the value_sequence:

33.1/5
```
[parallel for Item of X => Item]
```

Bounded (Run-Time) Errors

34/5 {*AI12-0262-1*} {*AI12-0348-1*} For a parallel reduction expression, it is a bounded error if the reducer subprogram is not associative. That is, for any arbitrary values of subtype *Value_Type A*, *B*, *C* and a reducer function *R*, the result of *R* (*A*, *R* (*B*, *C*)) should produce a result equal to *R* (*R* (*A*, *B*), *C*)); it is a bounded error if *R* does not. The possible consequences are Program_Error, or a result that does not match the equivalent sequential reduction expression due to the order of calls on the reducer subprogram being unspecified in the overall reduction. Analogous rules apply in the case of a reduction procedure.

Reason: In a sequential reduction expression, the reducer subprogram is called in a left-to-right order, whereas in a parallel reduction expression, the reducer subprogram is called in an order that depends on the number of logical threads of control that execute the reduction and on the elements/components given to each chunk. If the reducer is associative, this order does not matter, but in other cases, very different results are possible. While one can specify the *maximum* number of chunks, the actual number of chunks is unspecified. Similarly, the split of elements has only weak requirements. Thus, to get a consistent and portable result, an associative reducer is required for a parallel reduction. We define this as a Bounded (Run-Time) Errors to provide a stern warning about the required nature of the reducer subprogram and to let compilers detect the problem when possible. 34.a/5

To be honest: In this rule, “equal” means semantically equal. We don't care if the bit patterns differ but that the results mean the same thing. In particular, if the primitive equal is user-defined, that equality would be the one used to determine if this rule is violated. 34.b/5

Examples

{AI12-0262-1} {AI12-0429-1} Example of an expression function that returns its result as a reduction expression: 35/5

```
function Factorial(N : Natural) return Natural is
   ([for J in 1..N => J]'Reduce("*", 1));
```
36/5

{AI12-0262-1} {AI12-0429-1} Example of a reduction expression that computes the Sine of X using a Taylor expansion: 37/5

```
function Sine (X : Float; Num_Terms : Positive := 5) return Float is
   ([for I in 1..Num_Terms =>
      (-1.0)**(I-1) * X**(2*I-1)/Float(Factorial(2*I-1))]
        'Reduce("+", 0.0));
```
38/5

{AI12-0262-1} {AI12-0379-1} {AI12-0429-1} Example of a reduction expression that outputs the sum of squares: 39/5

```
Put_Line ("Sum of Squares is" &
          Integer'Image([for I in 1 .. 10 => I**2]'Reduce("+", 0)));
```
40/5

{AI12-0262-1} {AI12-0379-1} {AI12-0429-1} Example of a reduction expression used to compute the value of Pi: 41/5

```
--  See 3.5.7.
function Pi (Number_Of_Steps : Natural := 10_000) return Real is
  (1.0 / Real (Number_Of_Steps) *
    [for I in 1 .. Number_Of_Steps =>
        (4.0 / (1.0 + ((Real (I) - 0.5) *
           (1.0 / Real (Number_Of_Steps)))**2))]
              'Reduce("+", 0.0));
```
42/5

{AI12-0242-1} {AI12-0429-1} Example of a reduction expression used to calculate the sum of elements of an array of integers: 43/5

```
A'Reduce("+",0)   -- See 4.3.3.
```
44/5

{AI12-0242-1} {AI12-0429-1} Example of a reduction expression used to determine if all elements in a two-dimensional array of booleans are set to true: 45/5

```
Grid'Reduce("and", True)   -- See 3.6.
```
46/5

{AI12-0242-1} {AI12-0429-1} Example of a reduction expression used to calculate the minimum value of an array of integers in parallel: 47/5

```
A'Parallel_Reduce(Integer'Min, Integer'Last)
```
48/5

{AI12-0312-1} {AI12-0429-1} Example of a parallel reduction expression used to calculate the mean of the elements of a two-dimensional array of subtype Matrix (see 3.6) that are greater than 100.0: 49/5

```
type Accumulator is record
   Sum   : Real; -- See 3.5.7.
   Count : Integer;
end record;
```
50/5

51/5
```
function Accumulate (L, R : Accumulator) return Accumulator is
  (Sum   => L.Sum   + R.Sum,
   Count => L.Count + R.Count);
```

52/5
```
function Average_of_Values_Greater_Than_100 (M : Matrix) return Real is
   (declare
       Acc : constant Accumulator :=
          [parallel for Val of M when Val > 100.0 => (Val, 1)]
             'Reduce(Accumulate, (Sum => 0, Count => 0));
    begin
       Acc.Sum / Real(Acc.Count));
```

Extensions to Ada 2012

52.a/5 {*AI12-0005-1*} {*AI12-0242-1*} {*AI12-0262-1*} {*AI12-0348-1*} Reduction expressions and the associated attributes are new.

4.6 Type Conversions

1/3 {*AI05-0299-1*} [Explicit type conversions, both value conversions and view conversions, are allowed between closely related types as defined below. This subclause also defines rules for value and view conversions to a particular subtype of a type, both explicit ones and those implicit in other constructs.]

Syntax

2
```
type_conversion ::=
   subtype_mark(expression)
 | subtype_mark(name)
```

3 The *target subtype* of a type_conversion is the subtype denoted by the subtype_mark. The *operand* of a type_conversion is the expression or name within the parentheses; its type is the *operand type*.

4/3 {*AI05-0299-1*} One type is *convertible* to a second type if a type_conversion with the first type as operand type and the second type as target type is legal according to the rules of this subclause. Two types are convertible if each is convertible to the other.

4.a **Ramification:** Note that "convertible" is defined in terms of legality of the conversion. Whether the conversion would raise an exception at run time is irrelevant to this definition.

5/5 {*8652/0017*} {*AI95-00184-01*} {*AI95-00330-01*} {*AI12-0392-1*} A type_conversion is called a *view conversion* if both its target type and operand type are tagged, or if it appears in a call as an actual parameter of mode **out** or **in out**; other type_conversions are called *value conversions*.

5.a **Ramification:** A view conversion to a tagged type can appear in any context that requires an object name, including in an object renaming, the prefix of a selected_component, and if the operand is a variable, on the left side of an assignment_statement. View conversions to other types only occur as actual parameters. Allowing view conversions of untagged types in all contexts seemed to incur an undue implementation burden.

5.b/2 {*AI95-00330-01*} A type conversion appearing as an **in out** parameter in a generic instantiation is not a view conversion; the second part of the rule only applies to subprogram calls, not instantiations.

Name Resolution Rules

6 The operand of a type_conversion is expected to be of any type.

6.a **Discussion:** This replaces the "must be determinable" wording of Ada 83. This is equivalent to (but hopefully more intuitive than) saying that the operand of a type_conversion is a "complete context".

7 The operand of a view conversion is interpreted only as a name; the operand of a value conversion is interpreted as an expression.

7.a/5 **Reason:** {*AI12-0005-1*} This formally resolves the syntactic ambiguity between the two forms of type_conversion. This matters because an expression that is a name is evaluated and represents a value while a name by itself can be an object; we want a view conversion to be an object.

Ramification: {*AI12-0005-1*} This wording uses "interpreted as" rather than "shall be" so that this rule is not used to resolve overloading; it is solely about evaluation as described above. 7.b/4

Legality Rules

{*AI95-00251-01*} In a view conversion for an untagged type, the target type shall be convertible (back) to the operand type. 8/2

Reason: Untagged view conversions appear only as [**in**] **out** parameters. Hence, the reverse conversion must be legal as well. The forward conversion must be legal even for an **out** parameter, because (for example) actual parameters of an access type are always copied in anyway. 8.a/2

Paragraphs 9 through 20 were reorganized and moved below.

Discussion: {*AI95-00251-01*} The entire Legality Rules section has been reorganized to eliminate an unintentional incompatibility with Ada 83. In rare cases, a type conversion between two types related by derivation is not allowed by Ada 95, while it is allowed in Ada 83. The reorganization fixes this. Much of the wording of the legality section is unchanged, but it is reordered and reformatted. Because of the limitations of our tools, we had to delete and replace nearly the entire section. The text of Ada 95 paragraphs 8 through 12, 14, 15, 17, 19, 20, and 24 are unchanged (just moved); these are now 24.1 through 24.5, 24.12, 24.13, 24.17, 24.19, 24.20, and 8. 8.b/2

{*AI95-00251-01*} {*AI05-0115-1*} If there is a type (other than a root numeric type) that is an ancestor of both the target type and the operand type, or both types are class-wide types, then at least one of the following rules shall apply: 21/3

- {*AI95-00251-01*} The target type shall be untagged; or 21.1/2
- The operand type shall be covered by or descended from the target type; or 22

  Ramification: This is a conversion toward the root, which is always safe. 22.a
- {*AI95-00251-01*} The operand type shall be a class-wide type that covers the target type; or 23/2

  Ramification: This is a conversion of a class-wide type toward the leaves, which requires a tag check. See Dynamic Semantics. 23.a

  {*AI95-00251-01*} These two rules imply that a conversion from an ancestor type to a type extension is not permitted, as this would require specifying the values for additional components, in general, and changing the tag. An extension_aggregate has to be used instead, constructing a new value, rather than converting an existing value. However, a conversion from the class-wide type rooted at an ancestor type is permitted; such a conversion just verifies that the operand's tag is a descendant of the target. 23.b/2
- {*AI95-00251-01*} The operand and target types shall both be class-wide types and the specific type associated with at least one of them shall be an interface type. 23.1/2

  Ramification: We allow converting any class-wide type T'Class to or from a class-wide interface type even if the specific type T does not have an appropriate interface ancestor, because some extension of T might have the needed ancestor. This is similar to a conversion of a class-wide type toward the leaves of the tree, and we need to be consistent. Of course, there is a runtime check that the actual object has the needed interface. 23.c/2

{*AI95-00251-01*} {*AI05-0115-1*} If there is no type (other than a root numeric type) that is the ancestor of both the target type and the operand type, and they are not both class-wide types, one of the following rules shall apply: 24/3

- {*AI95-00251-01*} If the target type is a numeric type, then the operand type shall be a numeric type. 24.1/2
- {*AI95-00251-01*} If the target type is an array type, then the operand type shall be an array type. Further: 24.2/2
  - {*AI95-00251-01*} The types shall have the same dimensionality; 24.3/2
  - {*AI95-00251-01*} Corresponding index types shall be convertible; 24.4/2
  - {*AI95-00251-01*} The component subtypes shall statically match; 24.5/2
  - {*AI95-00392-01*} If the component types are anonymous access types, then the accessibility level of the operand type shall not be statically deeper than that of the target type; 24.6/2

24.b/2 **Reason:** For unrelated array types, the component types could have different accessibility, and we had better not allow a conversion of a local type into a global type, in case the local type points at local objects. We don't need a check for other types of components; such components necessarily are for related types, and either have the same accessibility or (for access discriminants) cannot be changed so the discriminant check will prevent problems.

24.7/2 - {*AI95-00246-01*} Neither the target type nor the operand type shall be limited;

24.c/2 **Reason:** We cannot allow conversions between unrelated limited types, as they may have different representations, and (since the types are limited), a copy cannot be made to reconcile the representations.

24.8/2 - {*AI95-00251-01*} {*AI95-00363-01*} If the target type of a view conversion has aliased components, then so shall the operand type; and

24.d/2 **Reason:** {*AI95-00363-01*} We cannot allow a view conversion from an object with unaliased components to an object with aliased components, because that would effectively allow pointers to unaliased components. This rule was missing from Ada 95.

24.9/2 - {*AI95-00246-01*} {*AI95-00251-01*} The operand type of a view conversion shall not have a tagged, private, or volatile subcomponent.

24.e/2 **Reason:** {*AI95-00246-01*} We cannot allow view conversions between unrelated might-be-by-reference types, as they may have different representations, and a copy cannot be made to reconcile the representations.

24.f/2 **Ramification:** These rules only apply to unrelated array conversions; different (weaker) rules apply to conversions between related types.

24.10/2 - {*AI95-00230-01*} If the target type is *universal_access*, then the operand type shall be an access type.

24.g/2 **Discussion:** Such a conversion cannot be written explicitly, of course, but it can be implicit (see below).

24.11/2 - {*AI95-00230-01*} {*AI95-00251-01*} If the target type is a general access-to-object type, then the operand type shall be *universal_access* or an access-to-object type. Further, if the operand type is not *universal_access*:

24.h/2 **Discussion:** The Legality Rules and Dynamic Semantics are worded so that a type_conversion T(X) (where T is an access type) is (almost) equivalent to the attribute_reference X.**all**'Access, where the result is of type T. The only difference is that the type_conversion accepts a null value, whereas the attribute_reference would raise Constraint_Error.

24.12/2 - {*AI95-00251-01*} If the target type is an access-to-variable type, then the operand type shall be an access-to-variable type;

24.i/2 **Ramification:** If the target type is an access-to-constant type, then the operand type can be access-to-constant or access-to-variable.

24.13/2 - {*AI95-00251-01*} If the target designated type is tagged, then the operand designated type shall be convertible to the target designated type;

24.14/2 - {*AI95-00251-01*} {*AI95-00363-01*} If the target designated type is not tagged, then the designated types shall be the same, and either:

24.15/2 - {*AI95-00363-01*} the designated subtypes shall statically match; or

24.16/2 - {*AI95-00363-01*} {*AI95-00384-01*} the designated type shall be discriminated in its full view and unconstrained in any partial view, and one of the designated subtypes shall be unconstrained;

24.j/2 **Ramification:** {*AI95-00363-01*} This does not require that types have a partial view in order to allow the conversion, simply that any partial view that does exist is unconstrained.

24.k/2 {*AI95-00384-01*} This allows conversions both ways (either subtype can be unconstrained); while Ada 95 only allowed the conversion if the target subtype is unconstrained. We generally want type conversions to be symmetric; which type is the target shouldn't matter for legality.

24.l/2 **Reason:** {*AI95-00363-01*} If the visible partial view is constrained, we do not allow conversion between unconstrained and constrained subtypes. This means that whether the full type had discriminants is not visible to clients of the partial view.

24.l.1/5 **Discussion:** {*AI12-0095-1*} {*AI12-0005-1*} We assume the worst in a generic body about whether or not a formal subtype has a constrained partial view; specifically, in a generic body a discriminated subtype is considered to have a

constrained partial view if it is a descendant of an untagged generic formal private or derived type (see 12.5.1 for the formal definition of this rule).

Reason: These rules are designed to ensure that aliased array objects only *need* "dope" if their nominal subtype is unconstrained, but they can always *have* dope if required by the run-time model (since no sliding is permitted as part of access type conversion). By contrast, aliased discriminated objects will always *need* their discriminants stored with them, even if nominally constrained. (Here, we are assuming an implementation that represents an access value as a single pointer.) 24.m/2

- {*AI95-00251-01*} {*AI05-0148-1*} {*AI05-0248-1*} {*AI12-0027-1*} The accessibility level of the operand type shall not be statically deeper than that of the target type, unless the target type is an anonymous access type of a stand-alone object. If the target type is that of such a stand-alone object, the accessibility level of the operand type shall not be statically deeper than that of the declaration of the stand-alone object. 24.17/4

  Ramification: {*AI05-0148-1*} The access parameter case is handled by a run-time check. runtime checks are also done in instance bodies, and for stand-alone objects of anonymous access types. 24.n/3

  Reason: We prohibit storing accesses to objects deeper than a stand-alone object of an anonymous access-to-object (even while we allow storing all other accesses) in order to prevent dangling accesses. 24.n.1/3

- {*AI95-00230-01*} If the target type is a pool-specific access-to-object type, then the operand type shall be *universal_access*. 24.18/2

  Reason: This allows **null** to be converted to pool-specific types. Without it, **null** could be converted to general access types but not pool-specific ones, which would be too inconsistent. Remember that these rules only apply to unrelated types, so we don't have to talk about conversions to derived or other related types. 24.o/2

- {*AI95-00230-01*} {*AI95-00251-01*} If the target type is an access-to-subprogram type, then the operand type shall be *universal_access* or an access-to-subprogram type. Further, if the operand type is not *universal_access*: 24.19/2

  - {*AI95-00251-01*} {*AI05-0239-1*} The designated profiles shall be subtype conformant. 24.20/3
  - {*AI95-00251-01*} {*AI12-0027-1*} The accessibility level of the operand type shall not be statically deeper than that of the target type. If the operand type is declared within a generic body, the target type shall be declared within the generic body. 24.21/4

    Reason: The reason it is illegal to convert from an access-to-subprogram type declared in a generic body to one declared outside that body is that in an implementation that shares generic bodies, subprograms declared inside the generic need to have a different calling convention — they need an extra parameter pointing to the data declared in the current instance. For subprograms declared in the spec, that's OK, because the compiler can know about them at compile time of the instantiation. 24.p/2

  - {*AI12-0380-1*} If the target type has a Global aspect other than **in out all** or Unspecified, then each mode of the Global aspect of the operand type shall identify a subset of the variables identified by the corresponding mode of the target type Global aspect, or by the **in out** mode of the target type Global aspect. 24.22/5
  - {*AI12-0064-2*} If the target type is nonblocking, the operand type shall be nonblocking. 24.23/5

{*AI12-0027-1*} In addition to the places where Legality Rules normally apply (see 12.3), these rules apply also in the private part of an instance of a generic unit. 24.24/5

Discussion: {*AI12-0005-1*} This applies to *all* of the Legality Rules in this section. It won't matter for the majority of these rules, but in any case that it does, we want to apply the same rules in the private part. (Ada got the default wrong for these, as there is only one known case where we don't want to recheck in the private part, see derivations without record extensions in 3.4.) 24.q/5

Static Semantics

A type_conversion that is a value conversion denotes the value that is the result of converting the value of the operand to the target subtype. 25

26/3 {*AI05-0264-1*} A type_conversion that is a view conversion denotes a view of the object denoted by the operand. This view is a variable of the target type if the operand denotes a variable; otherwise, it is a constant of the target type.

27 The nominal subtype of a type_conversion is its target subtype.

Dynamic Semantics

28 For the evaluation of a type_conversion that is a value conversion, the operand is evaluated, and then the value of the operand is *converted* to a *corresponding* value of the target type, if any. If there is no value of the target type that corresponds to the operand value, Constraint_Error is raised[; this can only happen on conversion to a modular type, and only when the operand value is outside the base range of the modular type.] Additional rules follow:

29 • Numeric Type Conversion

30 • If the target and the operand types are both integer types, then the result is the value of the target type that corresponds to the same mathematical integer as the operand.

31 • If the target type is a decimal fixed point type, then the result is truncated (toward 0) if the value of the operand is not a multiple of the *small* of the target type.

32 • If the target type is some other real type, then the result is within the accuracy of the target type (see G.2, "Numeric Performance Requirements", for implementations that support the Numerics Annex).

32.a **Discussion:** An integer type might have more bits of precision than a real type, so on conversion (of a large integer), some precision might be lost.

33 • If the target type is an integer type and the operand type is real, the result is rounded to the nearest integer (away from zero if exactly halfway between two integers).

33.a/2 **Discussion:** {*AI95-00267-01*} This was implementation defined in Ada 83. There seems no reason to preserve the nonportability in Ada 95. Round-away-from-zero is the conventional definition of rounding, and standard Fortran and COBOL both specify rounding away from zero, so for interoperability, it seems important to pick this. This is also the most easily "undone" by hand. Round-to-nearest-even is an alternative, but that is quite complicated if not supported by the hardware. In any case, this operation is not usually part of an inner loop, so predictability and portability are judged most important. A floating point attribute function Unbiased_Rounding is provided (see A.5.3) for those applications that require round-to-nearest-even, and a floating point attribute function Machine_Rounding (also see A.5.3) is provided for those applications that require the highest possible performance. "Deterministic" rounding is required for static conversions to integer as well. See 4.9.

34 • Enumeration Type Conversion

35 • The result is the value of the target type with the same position number as that of the operand value.

36 • Array Type Conversion

37 • If the target subtype is a constrained array subtype, then a check is made that the length of each dimension of the value of the operand equals the length of the corresponding dimension of the target subtype. The bounds of the result are those of the target subtype.

38 • If the target subtype is an unconstrained array subtype, then the bounds of the result are obtained by converting each bound of the value of the operand to the corresponding index type of the target type. For each nonnull index range, a check is made that the bounds of the range belong to the corresponding index subtype.

38.a **Discussion:** Only nonnull index ranges are checked, per AI83-00313.

39 • In either array case, the value of each component of the result is that of the matching component of the operand value (see 4.5.2).

39.a **Ramification:** This applies whether or not the component is initialized.

- {*AI95-00392-01*} If the component types of the array types are anonymous access types, then a check is made that the accessibility level of the operand type is not deeper than that of the target type. 39.1/2

  Reason: {*AI12-0005-1*} This check is needed for operands that are in instance bodies. Other cases are handled by the legality rule given previously. 39.b/5

- Composite (Non-Array) Type Conversion 40
  - The value of each nondiscriminant component of the result is that of the matching component of the operand value. 41

    Ramification: This applies whether or not the component is initialized. 41.a
  - [The tag of the result is that of the operand.] If the operand type is class-wide, a check is made that the tag of the operand identifies a (specific) type that is covered by or descended from the target type. 42

    Ramification: This check is certain to succeed if the operand type is itself covered by or descended from the target type. 42.a

    Proof: The fact that a type_conversion preserves the tag is stated officially in 3.9, “Tagged Types and Type Extensions” 42.b
  - For each discriminant of the target type that corresponds to a discriminant of the operand type, its value is that of the corresponding discriminant of the operand value; if it corresponds to more than one discriminant of the operand type, a check is made that all these discriminants are equal in the operand value. 43
  - For each discriminant of the target type that corresponds to a discriminant that is specified by the derived_type_definition for some ancestor of the operand type (or if class-wide, some ancestor of the specific type identified by the tag of the operand), its value in the result is that specified by the derived_type_definition. 44

    Ramification: It is a ramification of the rules for the discriminants of derived types that each discriminant of the result is covered either by this paragraph or the previous one. See 3.7. 44.a
  - For each discriminant of the operand type that corresponds to a discriminant that is specified by the derived_type_definition for some ancestor of the target type, a check is made that in the operand value it equals the value specified for it. 45
  - For each discriminant of the result, a check is made that its value belongs to its subtype. 46
- Access Type Conversion 47
  - {*AI05-0148-1*} {*AI05-0248-1*} For an access-to-object type, a check is made that the accessibility level of the operand type is not deeper than that of the target type, unless the target type is an anonymous access type of a stand-alone object. If the target type is that of such a stand-alone object, a check is made that the accessibility level of the operand type is not deeper than that of the declaration of the stand-alone object[; then if the check succeeds, the accessibility level of the target type becomes that of the operand type]. 48/3

    Ramification: {*AI05-0148-1*} This check is needed for operands that are access parameters, for stand-alone anonymous access objects, and in instance bodies. 48.a/3

    Note that this check can never fail for the implicit conversion to the anonymous type of an access parameter that is done when calling a subprogram with an access parameter. 48.b
  - {*AI95-00230-01*} {*AI95-00231-01*} If the operand value is null, the result of the conversion is the null value of the target type. 49/2

    Ramification: A conversion to an anonymous access type happens implicitly as part of initializing or assigning to an anonymous access object. 49.a/2
  - If the operand value is not null, then the result designates the same object (or subprogram) as is designated by the operand value, but viewed as being of the target designated subtype (or 50

profile); any checks associated with evaluating a conversion to the target designated subtype are performed.

50.a **Ramification:** The checks are certain to succeed if the target and operand designated subtypes statically match.

51/5 {*AI95-00231-01*} {*AI05-0153-3*} {*AI05-0290-1*} {*AI12-0071-1*} {*AI12-0333-1*} After conversion of the value to the target type, if the target subtype is constrained, a check is performed that the value satisfies this constraint. If the target subtype excludes null, then a check is made that the value is not null. If predicate checks are enabled for the target subtype (see 3.2.4), a check is performed that the value satisfies the predicates of the target subtype, unless the conversion is:.

51.a/2 **Ramification:** {*AI95-00231-01*} The first check above is a Range_Check for scalar subtypes, a Discriminant_Check or Index_Check for access subtypes, and a Discriminant_Check for discriminated subtypes. The Length_Check for an array conversion is performed as part of the conversion to the target type. The check for exclusion of null is an Access_Check.

51.1/5 - {*AI12-0333-1*} {*AI12-0432-1*} a view conversion that is the target of an assignment statement and is not referenced with a target_name, or an actual parameter of mode **out**; or

51.2/5 - {*AI12-0333-1*} an implicit subtype conversion of an actual parameter of mode **out** to the nominal subtype of its formal parameter.

51.b/5 **Ramification:** {*AI12-0333-1*} The reverse conversion applied to by-copy **out** parameters is *not* a view conversion and it is to the nominal subtype of the *actual* parameter, therefore any enabled predicate checks *are* performed.

52 For the evaluation of a view conversion, the operand name is evaluated, and a new view of the object denoted by the operand is created, whose type is the target type; if the target type is composite, checks are performed as above for a value conversion.

53 The properties of this new view are as follows:

54/1 - {*8652/0017*} {*AI95-00184-01*} If the target type is composite, the bounds or discriminants (if any) of the view are as defined above for a value conversion; each nondiscriminant component of the view denotes the matching component of the operand object; the subtype of the view is constrained if either the target subtype or the operand object is constrained, or if the target subtype is indefinite, or if the operand type is a descendant of the target type and has discriminants that were not inherited from the target type;

55/5 - {*AI12-0439-1*} If the target type is tagged, then an assignment to the view assigns to the corresponding part of the object denoted by the operand; otherwise, an assignment to the view assigns to the object, after converting the assigned value to the subtype of the object (which can raise Constraint_Error);

56/5 - {*AI12-0074-1*} {*AI12-0439-1*} Reading the value of the view yields the result of converting the value of the operand object to the target subtype (which can raise Constraint_Error), except if the object is of an elementary type and the view conversion is passed as an **out** parameter; in this latter case, the value of the operand object may be used to initialize the formal parameter without checking against any constraint of the target subtype (as described more precisely in 6.4.1).

56.a/4 **Reason:** {*AI12-0074-1*} This ensures that even an **out** parameter of an elementary type is initialized reasonably.

57/5 {*AI05-0290-1*} {*AI12-0096-1*} {*AI12-0449-1*} If an Accessibility_Check fails, Program_Error is raised. If a predicate check fails, the effect is as defined in 3.2.4. Any other check associated with a conversion raises Constraint_Error if it fails.

58 Conversion to a type is the same as conversion to an unconstrained subtype of the type.

58.a **Reason:** This definition is needed because the semantics of various constructs involves converting to a type, whereas an explicit type_conversion actually converts to a subtype. For example, the evaluation of a range is defined to convert the values of the expressions to the type of the range.

58.b **Ramification:** A conversion to a scalar type, or, equivalently, to an unconstrained scalar subtype, can raise Constraint_Error if the value is outside the base range of the type.

{*AI12-0027-1*} {*AI12-0226-1*} Evaluation of a value conversion of an object either creates a new anonymous object[(similar to the object created by the evaluation of an aggregate or a function call)] or yields a new view of the operand object without creating a new object: 58.1/5

- If the target type is a by-reference type and there is a type that is an ancestor of both the target type and the operand type then no new object is created; 58.2/4
- If the target type is an array type having aliased components and the operand type is an array type having unaliased components, then a new object is created; 58.3/4
- {*AI12-0226-1*} If the target type is an elementary type, then a new object is created; 58.4/5
- Otherwise, it is unspecified whether a new object is created. 58.5/5

{*AI12-0027-1*} If a new object is created, then the initialization of that object is an assignment operation. 58.6/5

Reason: This makes a difference in the case of converting from an array type with unaliased components to one with aliased components if the element type has a controlled part. 58.c/4

{*AI12-0226-1*} For an elementary type, the representation might change so we require a new object to avoid problems. 58.d/5

Implementation Note: {*AI12-0226-1*} The temporary object need not be materialized in most cases; it should be handled like the return object of a predefined operator. Generally, whether the object exists can only be detected if it is renamed (unless a part of the type is controlled). 58.e/5

Discussion: {*AI12-0226-1*} This set of rules does not apply in those cases where the operand is not an object (such as a value conversion of a named number); in such cases, the result isn't an object, so it isn't necessary to describe what that means. The rules cover all value conversions of composite types (since there aren't any values of composite types separate from objects). 58.f/5

NOTE 1 In addition to explicit type_conversions, type conversions are performed implicitly in situations where the expected type and the actual type of a construct differ, as is permitted by the type resolution rules (see 8.6). For example, an integer literal is of the type *universal_integer*, and is implicitly converted when assigned to a target of some specific integer type. Similarly, an actual parameter of a specific tagged type is implicitly converted when the corresponding formal parameter is of a class-wide type. 59

Even when the expected and actual types are the same, implicit subtype conversions are performed to adjust the array bounds (if any) of an operand to match the desired target subtype, or to raise Constraint_Error if the (possibly adjusted) value does not satisfy the constraints of the target subtype. 60

NOTE 2 {*AI95-00230-01*} A ramification of the overload resolution rules is that the operand of an (explicit) type_conversion cannot be an allocator, an aggregate, a string_literal, a character_literal, or an attribute_reference for an Access or Unchecked_Access attribute. Similarly, such an expression enclosed by parentheses is not allowed. A qualified_expression (see 4.7) can be used instead of such a type_conversion. 61/2

NOTE 3 The constraint of the target subtype has no effect for a type_conversion of an elementary type passed as an **out** parameter. Hence, it is recommended that the first subtype be specified as the target to minimize confusion (a similar recommendation applies to renaming and generic formal **in out** objects). 62

Examples

Examples of numeric type conversion: 63

```
Real(2*J)       -- value is converted to floating point
Integer(1.6)    -- value is 2
Integer(-0.4)   -- value is 0
```
64

Example of conversion between derived types: 65

```
type A_Form is new B_Form;
```
66

```
X : A_Form;
Y : B_Form;
```
67

```
X := A_Form(Y);
Y := B_Form(X);   -- the reverse conversion
```
68

Examples of conversions between array types: 69

```
type Sequence is array (Integer range <>) of Integer;
subtype Dozen is Sequence(1 .. 12);
Ledger : array(1 .. 100) of Integer;
```
70

71
```
Sequence(Ledger)              -- bounds are those of Ledger
Sequence(Ledger(31 .. 42))    -- bounds are 31 and 42
Dozen(Ledger(31 .. 42))       -- bounds are those of Dozen
```

Incompatibilities With Ada 83

71.a A character_literal is not allowed as the operand of a type_conversion, since there are now two character types in package Standard.

71.b The component subtypes have to statically match in an array conversion, rather than being checked for matching constraints at run time.

71.c Because sliding of array bounds is now provided for operations where it was not in Ada 83, programs that used to raise Constraint_Error might now continue executing and produce a reasonable result. This is likely to fix more bugs than it creates.

Extensions to Ada 83

71.d A type_conversion is considered the name of an object in certain circumstances (such a type_conversion is called a view conversion). In particular, as in Ada 83, a type_conversion can appear as an **in out** or **out** actual parameter. In addition, if the target type is tagged and the operand is the name of an object, then so is the type_conversion, and it can be used as the prefix to a selected_component, in an object_renaming_declaration, etc.

71.e We no longer require type-mark conformance between a parameter of the form of a type conversion, and the corresponding formal parameter. This had caused some problems for inherited subprograms (since there isn't really a type-mark for converted formals), as well as for renamings, formal subprograms, etc. See AI83-00245, AI83-00318, AI83-00547.

71.f We now specify “deterministic” rounding from real to integer types when the value of the operand is exactly between two integers (rounding is away from zero in this case).

71.g “Sliding” of array bounds (which is part of conversion to an array subtype) is performed in more cases in Ada 95 than in Ada 83. Sliding is not performed on the operand of a membership test, nor on the operand of a qualified_expression. It wouldn't make sense on a membership test, and we wish to retain a connection between subtype membership and subtype qualification. In general, a subtype membership test returns True if and only if a corresponding subtype qualification succeeds without raising an exception. Other operations that take arrays perform sliding.

Wording Changes from Ada 83

71.h We no longer explicitly list the kinds of things that are not allowed as the operand of a type_conversion, except in a NOTE.

71.i/3 {*AI05-0299-1*} The rules in this subclause subsume the rules for "parameters of the form of a type conversion", and have been generalized to cover the use of a type conversion as a name.

Incompatibilities With Ada 95

71.j/2 {*AI95-00246-01*} **Amendment Correction:** Conversions between unrelated array types that are limited or (for view conversions) might be by-reference types are now illegal. The representations of two such arrays may differ, making the conversions impossible. We make the check here, because legality should not be based on representation properties. Such conversions are likely to be rare, anyway. There is a potential that this change would make a working program illegal (if the types have the same representation).

71.k/2 {*AI95-00363-01*} If a discriminated full type has a partial view (private type) that is constrained, we do not allow conversion between access-to-unconstrained and access-to-constrained subtypes designating the type. Ada 95 allowed this conversion and the declaration of various access subtypes, requiring that the designated object be constrained and thus making details of the implementation of the private type visible to the client of the private type. See 4.8 for more on this topic.

Extensions to Ada 95

71.l/2 {*AI95-00230-01*} Conversion rules for *universal_access* were defined. These allow the use of anonymous access values in equality tests (see 4.5.2), and also allow the use of **null** in type conversions and other contexts that do not provide a single expected type.

71.m/2 {*AI95-00384-01*} A type conversion from an access-to-discriminated and unconstrained object to an access-to-discriminated and constrained one is allowed. Ada 95 only allowed the reverse conversion, which was weird and asymmetric. Of course, a constraint check will be performed for this conversion.

Wording Changes from Ada 95

{*8652/0017*} {*AI95-00184-01*} **Corrigendum:** Wording was added to ensure that view conversions are constrained, and that a tagged view conversion has a tagged object. Both rules are needed to avoid having a way to change the discriminants of a constrained object. 71.n/2

{*8652/0008*} {*AI95-00168-01*} **Corrigendum:** Wording was added to ensure that the aliased status of array components cannot change in a view conversion. This rule was needed to avoid having a way to change the discriminants of an aliased object. This rule was repealed later, as Ada 2005 allows changing the discriminants of an aliased object. 71.o/2

{*AI95-00231-01*} Wording was added to check subtypes that exclude null (see 3.10). 71.p/2

{*AI95-00251-01*} The organization of the legality rules was changed, both to make it clearer, and to eliminate an unintentional incompatibility with Ada 83. The old organization prevented type conversions between some types that were related by derivation (which Ada 83 always allowed). 71.q/2

{*AI95-00330-01*} {*AI05-0005-1*} Clarified that an untagged type conversion appearing as a generic actual parameter for a generic **in out** formal parameter is not a view conversion (and thus is illegal). This confirms the ACATS tests, so all implementations already follow this interpretation. 71.r/3

{*AI95-00363-01*} Rules added by the Corrigendum to eliminate problems with discriminants of aliased components changing were removed, as we now generally allow discriminants of aliased components to be changed. 71.s/2

{*AI95-00392-01*} Accessibility checks on conversions involving types with anonymous access components were added. These components have the level of the type, and conversions can be between types at different levels, which could cause dangling access values in the absence of such checks. 71.t/2

Inconsistencies With Ada 2005

{*AI05-0148-1*} A stand-alone object of an anonymous access-to-object type now has dynamic accessibility. Normally, this will make programs legal that were illegal in Ada 2005. However, it is possible that a program that previously raised Program_Error now will not. It is very unlikely that an existing program intentionally depends on the exception being raised; the change is more likely to fix bugs than introduce them. 71.u/3

Wording Changes from Ada 2005

{*AI05-0115-1*} **Correction:** Clarified that a root numeric type is not considered a common ancestor for a conversion. 71.v/3

{*AI05-0153-3*} {*AI05-0290-1*} Added rules so that predicate aspects (see 3.2.4) are enforced on subtype conversion. 71.w/3

Inconsistencies With Ada 2012

{*AI12-0333-1*} {*AI12-0432-1*} **Correction:** Predicate checks are no longer made on any **out** parameters before a call (they're still made when the call returns). This was already true for elementary **out** parameters. If a program depends on a predicate check failing on an inbound **out** composite parameter, it will get an incorrect result. Similarly, predicate checks are no longer made on the view conversion of a target of an assignment_statement. Both of these cases seem quite unlikely, as programs (outside of ACATS tests) that depend on the failure of checks are very rare, and the predicate might be checking uninitialized components (making check failure unreliable). 71.x/5

Incompatibilities With Ada 2012

{*AI12-0095-1*} {*AI12-0005-1*} **Corrigendum:** Because of a rule added in 12.5.1, the checks for the legality of an access type conversion in a generic body were strengthened to use an assume-the-worst rule. This case is rather unlikely because a formal private or derived type with discriminants is required along with a conversion between two access types whose designated types don't statically match, and any such programs were at risk having objects disappear while valid access values still pointed at them. 71.y/5

Wording Changes from Ada 2012

{*AI12-0027-1*} **Corrigendum:** Moved the generic boilerplate so that it covers all Legality Rules in this subclause. This was always intended, but it is not expected to change anything other than conversions between unrelated arrays. 71.z/4

{*AI12-0027-1*} **Corrigendum:** Added a formal definition of the copy potentially created by a value conversion of a composite type, so properties like finalization and accessibility are properly defined. This model was always intended and expected (else 13.6 would not work), but it was not previously formally defined. 71.aa/4

{*AI12-0071-1*} **Corrigendum:** Updated wording of type conversions to use the new term "satisfies the predicates" (see 3.2.4). 71.bb/4

{*AI12-0074-1*} **Corrigendum:** Clarified the wording describing the effect of view conversions of **out** parameters such that it is clear that the detailed effect is defined in 6.4.1, not here. 71.cc/4

71.dd/4 {*AI12-0096-1*} **Corrigendum:** Updated wording of type conversions so that the exception raise or other effect of a failed predicate check is as defined in 3.2.4; we don't want to repeat those rules here. This doesn't change the behavior for predicate checks possible in original Ada 2012, only ones using the new aspect Predicate_Failure.

71.ee/5 {*AI12-0064-2*} {*AI12-0380-1*} Required Global (see 6.1.2) and Nonblocking (see 9.5) aspect matching for access-to-subprogram conversions.

71.ff/5 {*AI12-0226-1*} Described the objects associated with value conversions of elementary types. This is necessary to support an extension documented in 3.3.

71.gg/5 {*AI12-0392-1*} **Correction:** Eliminated the requirement that the operand of a view conversion be a simple name. The requirement could cause unintended consequences in the case where the operand of a type conversion represents an object but is more complex than a simple name (such as a qualified_expression or conditional_expression).

4.7 Qualified Expressions

1 [A qualified_expression is used to state explicitly the type, and to verify the subtype, of an operand that is either an expression or an aggregate.]

Syntax

2
```
qualified_expression ::=
  subtype_mark'(expression) | subtype_mark'aggregate
```

Name Resolution Rules

3/5 {*AI12-0325-1*} The expected type for the *operand* (the expression or aggregate) is determined by the subtype_mark. Furthermore, the operand shall resolve to be either the specified expected type or a universal type that covers it.

3.a/5 **Reason:** {*AI12-0325-1*} The first sentence defines the expected type for rules that assume one is defined. The second sentence prevents the use of the various implicit conversions that are usually allowed for expected types (except the one for numeric literals). The intent is that a qualified expression is similar to an assertion about the subtype of the operand, and thus implicit conversions would interfere with that intent.

Static Semantics

3.1/3 {*AI05-0003-1*} [If the operand of a qualified_expression denotes an object, the qualified_expression denotes a constant view of that object.] The nominal subtype of a qualified_expression is the subtype denoted by the subtype_mark.

3.b/3 **Proof:** {*AI05-0003-1*} This is stated in 3.3.

Dynamic Semantics

4/5 {*AI12-0100-1*} {*AI12-0449-1*} The evaluation of a qualified_expression evaluates the operand (and if of a universal type, converts it to the type determined by the subtype_mark) and checks that its value belongs to the subtype denoted by the subtype_mark. The exception Constraint_Error is raised if this check fails. Furthermore, if predicate checks are enabled for the subtype denoted by the subtype_mark, a check is performed as defined in 3.2.4 that the value satifies the predicates of the subtype.

4.a **Ramification:** This is one of the few contexts in Ada 95 where implicit subtype conversion is not performed prior to a constraint check, and hence no “sliding” of array bounds is provided.

4.a.1/4 {*AI12-0100-1*} The effect of a failed predicate check is as defined in 3.2.4; such a check could raise any exception, not just Constraint_Error or Assertion_Error.

4.b **Reason:** Implicit subtype conversion is not provided because a qualified_expression with a constrained target subtype is essentially an assertion about the subtype of the operand, rather than a request for conversion. An explicit type_conversion can be used rather than a qualified_expression if subtype conversion is desired.

4.c/4 {*AI12-0100-1*} We do a predicate check here so that a qualified_expression never allows something that the equivalent type_conversion would not allow.

NOTE {*AI12-0440-1*} When a given context does not uniquely identify an expected type, a qualified_expression can be used to do so. In particular, if an overloaded name or aggregate is passed to an overloaded subprogram, it can be necessary to qualify the operand to resolve its type. 5/5

Examples

Examples of disambiguating expressions using qualification: 6

```
type Mask is (Fix, Dec, Exp, Signif);                                                    7
type Code is (Fix, Cla, Dec, Tnz, Sub);

Print (Mask'(Dec));  -- Dec is of type Mask                                              8
Print (Code'(Dec));  -- Dec is of type Code

{AI12-0442-1} for J in Code'(Fix) .. Code'(Dec) loop ... -- qualification is necessary for either Fix or Dec   9/5
for J in Code range Fix .. Dec loop ...      -- qualification unnecessary
for J in Code'(Fix) .. Dec loop ...          -- qualification unnecessary for Dec

Dozen'(1 | 3 | 5 | 7 => 2, others => 0) -- see 4.6                                      10
```

Wording Changes from Ada 2005

{*AI05-0003-1*} Added a definition of the nominal subtype of a qualified_expression. 10.a/3

Inconsistencies With Ada 2012

{*AI12-0100-1*} {*AI12-0005-1*} **Corrigendum:** A qualified_expression now performs a predicate check for the named subtype (if it is enabled). Original Ada 2012 did not include that check (an omission). While this is formally inconsistent (an exception could be raised when none would be raised by original Ada 2012), cases where this scenario arises are likely to be rare (the qualified expression would have to have a stricter subtype than the following usage) and the check is more likely to detect bugs than be unexpected. 10.b/5

Wording Changes from Ada 2012

{*AI12-0325-1*} Reworded the resolution rule so that the operand of a qualified_expression has an expected type. This eliminates an annoying inconsistency in the language definition. 10.c/5

4.8 Allocators

[The evaluation of an allocator creates an object and yields an access value that designates the object.] 1

Syntax

{*AI05-0111-3*} allocator ::= 2/3
new [subpool_specification] subtype_indication
| **new** [subpool_specification] qualified_expression

{*AI05-0111-3*} subpool_specification ::= (*subpool_handle*_name) 2.1/3

{*AI05-0104-1*} For an allocator with a subtype_indication, the subtype_indication shall not specify a null_exclusion. 2.2/3

Reason: Such an uninitialized allocator would necessarily raise Constraint_Error, as the default value is **null**. Also note that the syntax does not allow a null_exclusion in an initialized allocator, so it makes sense to make the uninitialized case illegal as well. 2.a/3

Name Resolution Rules

{*8652/0010*} {*AI95-00127-01*} {*AI05-0111-3*} {*AI05-0269-1*} The expected type for an allocator shall be a single access-to-object type with designated type *D* such that either *D* covers the type determined by the subtype_mark of the subtype_indication or qualified_expression, or the expected type is anonymous and the determined type is *D*'Class. A *subpool_handle*_name is expected to be of any type descended from Subpool_Handle, which is the type used to identify a subpool, declared in package System.Storage_Pools.Subpools (see 13.11.4). 3/3

3.a **Discussion:** See 8.6, “The Context of Overload Resolution” for the meaning of “shall be a single ... type whose ...”.

3.a.1/1 **Ramification:** {*8652/0010*} {*AI95-00127-01*} An allocator is allowed as a controlling parameter of a dispatching call (see 3.9.2).

Legality Rules

4 An *initialized* allocator is an allocator with a qualified_expression. An *uninitialized* allocator is one with a subtype_indication. In the subtype_indication of an uninitialized allocator, a constraint is permitted only if the subtype_mark denotes an [unconstrained] composite subtype; if there is no constraint, then the subtype_mark shall denote a definite subtype.

4.a **Ramification:** For example, ... **new** S'Class ... (with no initialization expression) is illegal, but ... **new** S'Class'(X) ... is legal, and takes its tag and constraints from the initial value X. (Note that the former case cannot have a constraint.)

5/2 {*AI95-00287-01*} If the type of the allocator is an access-to-constant type, the allocator shall be an initialized allocator.

5.a/2 *This paragraph was deleted.*{*AI95-00287-01*}

5.1/3 {*AI05-0111-3*} If a subpool_specification is given, the type of the storage pool of the access type shall be a descendant of Root_Storage_Pool_With_Subpools.

5.2/3 {*AI95-00344-01*} If the designated type of the type of the allocator is class-wide, the accessibility level of the type determined by the subtype_indication or qualified_expression shall not be statically deeper than that of the type of the allocator.

5.b/2 **Reason:** This prevents the allocated object from outliving its type.

5.3/3 {*AI95-00416-01*} {*AI05-0051-1*} If the subtype determined by the subtype_indication or qualified_expression of the allocator has one or more access discriminants, then the accessibility level of the anonymous access type of each access discriminant shall not be statically deeper than that of the type of the allocator (see 3.10.2).

5.c/2 **Reason:** This prevents the allocated object from outliving its discriminants.

5.4/3 {*AI95-00366-01*} {*AI05-0052-1*} {*AI05-0157-1*} An allocator shall not be of an access type for which the Storage_Size has been specified by a static expression with value zero or is defined by the language to be zero.

5.d/2 **Reason:** An allocator for an access type that has Storage_Size specified to be zero is required to raise Storage_Error anyway. It's better to detect the error at compile-time, as the allocator might be executed infrequently. This also simplifies the rules for Pure units, where we do not want to allow any allocators for library-level access types, as they would represent state.

5.e/3 {*AI05-0157-1*} We don't need a special rule to cover generic formals (unlike many other similar Legality Rules). There are only two cases of interest. For formal access types, the Storage_Size property is not known in the generic, and surely isn't static, so this Legality Rule can never apply. For a formal derived type, this Legality Rule can only be triggered by a parent type having one of the appropriate properties. But Storage_Size can never be specified for a derived access type, so it always has the same value for all child types; additionally, a type derived from a remote access type (which has Storage_Size defined to be zero) is also a remote access type. That means that any actual that would match the formal derived type necessarily has the same Storage_Size properties, so it is harmless (and preferable) to check them in the body - they are always known in that case. For other formal types,allocators are not allowed, so we don't need to consider them. So we don't need an assume-the-best rule here.

5.5/3 {*AI05-0052-1*} If the designated type of the type of the allocator is limited, then the allocator shall not be used to define the value of an access discriminant, unless the discriminated type is immutably limited (see 7.5).

5.f/3 **Reason:** Because coextensions work very much like parts, we don't want users creating limited coextensions for nonlimited types. This would be similar to extending a nonlimited type with a limited component. We check this on the allocator. Note that there is an asymmetry in what types are considered limited; this is required to preserve privacy. We have to assume that the designated type might be limited as soon as we see a limited partial view, but we want to ensure that the containing object is of a type that is always limited.

{*AI05-0052-1*} In addition to the places where Legality Rules normally apply (see 12.3), these rules apply also in the private part of an instance of a generic unit. 5.6/3

Discussion: This applies to all of the Legality Rules of this subclause. 5.g/3

Static Semantics

{*AI95-00363-01*} {*AI05-0041-1*} If the designated type of the type of the allocator is elementary, then the subtype of the created object is the designated subtype. If the designated type is composite, then the subtype of the created object is the designated subtype when the designated subtype is constrained or there is an ancestor of the designated type that has a constrained partial view; otherwise, the created object is constrained by its initial value [(even if the designated subtype is unconstrained with defaults)]. 6/3

Discussion: See AI83-00331. 6.a

Reason: {*AI95-00363-01*} All objects created by an allocator are aliased, and most aliased composite objects need to be constrained so that access subtypes work reasonably. Problematic access subtypes are prohibited for types with a constrained partial view. 6.b/2

Discussion: {*AI95-00363-01*} If there is a constrained partial view of the type, this allows the objects to be unconstrained. This eliminates privacy breaking (we don't want the objects to act differently simply because they're allocated). Such a created object is effectively constrained by its initial value if the access type is an access-to-constant type, or the designated type is limited (in all views), but we don't need to state that here. It is implicit in other rules. Note, however, that a value of an access-to-constant type can designate a variable object via 'Access or conversion, and the variable object might be assigned by some other access path, and that assignment might alter the discriminants. 6.c/2

Dynamic Semantics

{*AI95-00373-01*} For the evaluation of an initialized allocator, the evaluation of the qualified_expression is performed first. An object of the designated type is created and the value of the qualified_expression is converted to the designated subtype and assigned to the object. 7/2

Ramification: The conversion might raise Constraint_Error. 7.a

For the evaluation of an uninitialized allocator, the elaboration of the subtype_indication is performed first. Then: 8

- {*AI95-00373-01*} If the designated type is elementary, an object of the designated subtype is created and any implicit initial value is assigned; 9/2
- {*8652/0002*} {*AI95-00171-01*} {*AI95-00373-01*} If the designated type is composite, an object of the designated type is created with tag, if any, determined by the subtype_mark of the subtype_indication. This object is then initialized by default (see 3.3.1) using the subtype_indication to determine its nominal subtype. A check is made that the value of the object belongs to the designated subtype. Constraint_Error is raised if this check fails. This check and the initialization of the object are performed in an arbitrary order. 10/2

Discussion: AI83-00150. 10.a

{*AI95-00344-01*} {*AI95-00416-01*} {*AI05-0024-1*} {*AI05-0051-1*} {*AI05-0234-1*} For any allocator, if the designated type of the type of the allocator is class-wide, then a check is made that the master of the type determined by the subtype_indication, or by the tag of the value of the qualified_expression, includes the elaboration of the type of the allocator. If any part of the subtype determined by the subtype_indication or qualified_expression of the allocator (or by the tag of the value if the type of the qualified_expression is class-wide) has one or more access discriminants, then a check is made that the accessibility level of the anonymous access type of each access discriminant is not deeper than that of the type of the allocator. Program_Error is raised if either such check fails. 10.1/3

Reason: {*AI95-00344-01*} {*AI05-0024-1*} The master check on class-wide types prevents the allocated object from outliving its type. We need the run-time check in instance bodies, or when the type of the qualified_expression is class-wide (other cases are statically detected). 10.b/3

10.b.1/3 {*AI05-0024-1*} We can't use the normal accessibility level “deeper than” check here because we may have “incomparable” levels if the appropriate master and the type declaration belong to two different tasks. This can happen when checking the master of the tag for an allocator initialized by a parameter passed in to an accept statement, if the type of the allocator is an access type declared in the enclosing task body. For example:

10.b.2/3
```
task body TT is
   type Acc_TC is access T'Class;
   P : Acc_TC;
begin
   accept E(X : T'Class) do
      P := new T'Class'(X);
         -- Master check on tag of X.
         -- Can't use "accessibility levels" since they might be incomparable.
         -- Must revert to checking that the master of the type identified by
         -- X'tag includes the elaboration of Acc_TC, so it is sure to outlive it.
   end E;
```

10.c/2 {*AI95-00416-01*} The accessibility check on access discriminants prevents the allocated object from outliving its discriminants.

10.2/2 {*AI95-00280-01*} If the object to be created by an allocator has a controlled or protected part, and the finalization of the collection of the type of the allocator (see 7.6.1) has started, Program_Error is raised.

10.d/2 **Reason:** If the object has a controlled or protected part, its finalization is likely to be nontrivial. If the allocation was allowed, we could not know whether the finalization would actually be performed. That would be dangerous to otherwise safe abstractions, so we mandate a check here. On the other hand, if the finalization of the object will be trivial, we do not require (but allow) the check, as no real harm could come from late allocation.

10.e/2 **Discussion:** This check can only fail if an allocator is evaluated in code reached from a Finalize routine for a type declared in the same master. That's highly unlikely; Finalize routines are much more likely to be deallocating objects than allocating them.

10.3/2 {*AI95-00280-01*} If the object to be created by an allocator contains any tasks, and the master of the type of the allocator is completed, and all of the dependent tasks of the master are terminated (see 9.3), then Program_Error is raised.

10.f/2 **Reason:** A task created after waiting for tasks has finished could depend on freed data structures, and certainly would never be awaited.

10.4/3 {*AI05-0111-3*} If the allocator includes a *subpool_handle*_name, Constraint_Error is raised if the subpool handle is **null**. Program_Error is raised if the subpool does not *belong* (see 13.11.4) to the storage pool of the access type of the allocator.

10.g/3 **Implementation Note:** This can be implemented by comparing the result of Pool_of_Subpool to a reference to the storage pool object. Pool_of_Subpool's parameter is **not null**, so the check for null falls out naturally.

10.h/3 **Reason:** This detects cases where the subpool belongs to another pool, or to no pool at all. This includes detecting dangling subpool handles so long as the subpool object (the object designated by the handle) still exists. (If the subpool object has been deallocated, execution is erroneous; it is likely that this check will still detect the problem, but there cannot be a guarantee.)

11 [If the created object contains any tasks, they are activated (see 9.2).] Finally, an access value that designates the created object is returned.

Bounded (Run-Time) Errors

11.1/2 {*AI95-00280-01*} It is a bounded error if the finalization of the collection of the type (see 7.6.1) of the allocator has started. If the error is detected, Program_Error is raised. Otherwise, the allocation proceeds normally.

11.a/2 **Discussion:** This check is required in some cases; see above.

12 NOTE 1 Allocators cannot create objects of an abstract type. See 3.9.3.

13 NOTE 2 If any part of the created object is controlled, the initialization includes calls on corresponding Initialize or Adjust procedures. See 7.6.

14/5 NOTE 3 {*AI12-0440-1*} As explained in 13.11, “Storage Management”, the storage for an object allocated by an allocator comes from a storage pool (possibly user defined). The exception Storage_Error is raised by an allocator if there is not enough storage. Instances of Unchecked_Deallocation can be used to explicitly reclaim storage.

NOTE 4 {*AI05-0229-1*} {*AI12-0442-1*} Implementations can, if desired, provide garbage collection. 15/5

Ramification: Note that in an allocator, the exception Constraint_Error can be raised by the evaluation of the qualified_expression, by the elaboration of the subtype_indication, or by the initialization. 15.a

Discussion: By default, the implementation provides the storage pool. The user may exercise more control over storage management by associating a user-defined pool with an access type. 15.b

Examples

Examples of allocators: 16

```
new Cell'(0, null, null)                    -- initialized explicitly, see 3.10.1   17
new Cell'(Value => 0, Succ => null, Pred => null)
                                            -- initialized explicitly
new Cell                                    -- not initialized

new Matrix(1 .. 10, 1 .. 20)                -- the bounds only are given            18
new Matrix'(1 .. 10 => (1 .. 20 => 0.0))    -- initialized explicitly

new Buffer(100)                             -- the discriminant only is given       19
new Buffer'(Size => 80, Pos => 0, Value => (1 .. 80 => 'A'))
                                            -- initialized explicitly

Expr_Ptr'(new Literal)                                                              20
                    -- allocator for access-to-class-wide type, see 3.9.1
Expr_Ptr'(new Literal'(Expression with 3.5)) -- initialized explicitly
```

Incompatibilities With Ada 83

The subtype_indication of an uninitialized allocator may not have an explicit constraint if the designated type is an access type. In Ada 83, this was permitted even though the constraint had no effect on the subtype of the created object. 20.a/1

Extensions to Ada 83

Allocators creating objects of type *T* are now overloaded on access types designating *T*'Class and all class-wide types that cover *T*. 20.b

Implicit array subtype conversion (sliding) is now performed as part of an initialized allocator. 20.c

Wording Changes from Ada 83

We have used a new organization, inspired by the ACID document, that makes it clearer what is the subtype of the created object, and what subtype conversions take place. 20.d

Discussion of storage management issues, such as garbage collection and the raising of Storage_Error, has been moved to 13.11, “Storage Management”. 20.e

Inconsistencies With Ada 95

{*AI95-00363-01*} If the designated type has a constrained partial view, the allocated object can be unconstrained. This might cause the object to take up a different amount of memory, and might cause the operations to work where they previously would have raised Constraint_Error. It's unlikely that the latter would actually matter in a real program (Constraint_Error usually indicates a bug that would be fixed, not left in a program.) The former might cause Storage_Error to be raised at a different time than in an Ada 95 program. 20.f/2

Incompatibilities With Ada 95

{*AI95-00366-01*} An allocator for an access type that has Storage_Size specified to be zero is now illegal. Ada 95 allowed the allocator, but it had to raise Storage_Error if executed. The primary impact of this change should be to detect bugs. 20.g/2

Extensions to Ada 95

{*8652/0010*} {*AI95-00127-01*} **Corrigendum:** An allocator can be a controlling parameter of a dispatching call. This was an oversight in Ada 95. 20.h/2

{*AI95-00287-01*} Initialized allocators are allowed when the designated type is limited. 20.i/2

Wording Changes from Ada 95

20.j/2 {*8652/0002*} {*AI95-00171-01*} **Corrigendum:** Clarified the elaboration of per-object constraints for an uninitialized allocator.

20.k/2 {*AI95-00280-01*} Program_Error is now raised if the allocator occurs after the finalization of the collection or the waiting for tasks. This is not listed as an incompatibility as the Ada 95 behavior was unspecified, and Ada 95 implementations tend to generate programs that crash in this case.

20.l/2 {*AI95-00344-01*} Added accessibility checks to class-wide allocators. These checks could not fail in Ada 95 (as all of the designated types had to be declared at the same level, so the access type would necessarily have been at the same level or more nested than the type of allocated object).

20.m/2 {*AI95-00373-01*} Revised the description of evaluation of uninitialized allocators to use "initialized by default" so that the ordering requirements are the same for all kinds of objects that are default-initialized.

20.n/2 {*AI95-00416-01*} Added accessibility checks to access discriminants of allocators. These checks could not fail in Ada 95 as the discriminants always have the accessibility of the object.

Incompatibilities With Ada 2005

20.o/3 {*AI05-0052-1*} **Correction:** Added a rule to prevent limited coextensions of nonlimited types. Allowing this would have far-reaching implementation costs. Because of those costs, it seems unlikely that any implementation ever supported it properly and thus it is unlikely that any existing code depends on this capability.

20.p/3 {*AI05-0104-1*} **Correction:** Added a rule to make null_exclusions illegal for uninitialized allocators, as such an allocator would always raise Constraint_Error. Programs that depend on the unconditional raising of a predefined exception should be very rare.

Extensions to Ada 2005

20.q/3 {*AI05-0111-3*} Subpool handles (see 13.11.4) can be specified in an allocator.

Wording Changes from Ada 2005

20.r/3 {*AI05-0024-1*} **Correction:** Corrected the master check for tags since the masters may be for different tasks and thus incomparable.

20.s/3 {*AI05-0041-1*} **Correction:** Corrected the rules for when a designated object is constrained by its initial value so that types derived from a partial view are handled properly.

20.t/3 {*AI05-0051-1*} {*AI05-0234-1*} **Correction:** Corrected the accessibility check for access discriminants so that it does not depend on the designated type (which might not have discriminants when the allocated type does).

4.9 Static Expressions and Static Subtypes

1 Certain expressions of a scalar or string type are defined to be static. Similarly, certain discrete ranges are defined to be static, and certain scalar and string subtypes are defined to be static subtypes. [*Static* means determinable at compile time, using the declared properties or values of the program entities.]

1.a **Discussion:** As opposed to more elaborate data flow analysis, etc.

Language Design Principles

1.b For an expression to be static, it has to be calculable at compile time.

1.c Only scalar and string expressions are static.

1.d To be static, an expression cannot have any nonscalar, nonstring subexpressions (though it can have nonscalar constituent names). A static scalar expression cannot have any nonscalar subexpressions. There is one exception — a membership test for a string subtype can be static, and the result is scalar, even though a subexpression is nonscalar.

1.e The rules for evaluating static expressions are designed to maximize portability of static calculations.

1.f/5 **Reason:** {*AI12-0201-1*} We support static string expressions so that, for example, the aspect_definition for a Link_Name aspect can contain a concatenation. We don't support static aggregates (even for string types) or non-string static nonscalar types; we're trying to keep it cheap and simple (from the implementer's viewpoint).

Static Semantics

2 A static expression is [a scalar or string expression that is] one of the following:

- {*AI12-0249-1*} a numeric_literal of a numeric type; 3/5

  Ramification: {*AI12-0249-1*} A numeric_literal of a numeric type is always a static expression, even if its expected type is not that of a static subtype. However, if its value is explicitly converted to, or qualified by, a nonstatic subtype, the resulting expression is nonstatic. Non-numeric types can have numeric literals if aspect Integer_Literal or Real_Literal is used; these are never static. 3.a/5

- a string_literal of a static string subtype; 4

  Ramification: That is, the constrained subtype defined by the index range of the string is static. Note that elementary values don't generally have subtypes, while composite values do (since the bounds or discriminants are inherent in the value). 4.a

- {*AI12-0394-1*} a name that denotes the declaration of a static constant; 5/5

- {*AI12-0394-1*} a name that denotes a named number, and that is interpreted as a value of a numeric type; 5.1/5

  To be honest: {*AI12-0394-1*} This is referring the resolution of the named number and not the Static Semantics (for which all named numbers are values of a universal numeric type). The word “interpreted” is intended to make the distinction. 5.a.1/5

  Ramification: Note that enumeration literals are covered by the function_call case. 5.a

- a function_call whose *function*_name or *function*_prefix statically denotes a static function, and whose actual parameters, if any (whether given explicitly or by default), are all static expressions; 6

  Ramification: This includes uses of operators that are equivalent to function_calls. 6.a

- an attribute_reference that denotes a scalar value, and whose prefix denotes a static scalar subtype; 7

  Ramification: Note that this does not include the case of an attribute that is a function; a reference to such an attribute is not even an expression. See above for function *calls*. 7.a

  An implementation may define the staticness and other properties of implementation-defined attributes. 7.b

- {*AI12-0368-1*} an attribute_reference whose prefix statically names a statically constrained array object or array subtype, and whose attribute_designator is First, Last, or Length, with an optional dimension; 8/5

- {*AI12-0064-2*} an attribute_reference whose prefix denotes a non-generic entity that is not declared in a generic unit, and whose attribute_designator is Nonblocking; 8.1/5

- {*AI12-0201-1*} a type_conversion whose subtype_mark denotes a static [(scalar or string)] subtype, and whose operand is a static expression; 9/5

- a qualified_expression whose subtype_mark denotes a static [(scalar or string)] subtype, and whose operand is a static expression; 10

  Ramification: This rules out the subtype_mark'aggregate case. 10.a

  Reason: Adding qualification to an expression shouldn't make it nonstatic, even for strings. 10.b

- {*AI05-0158-1*} {*AI05-0269-1*} {*AI12-0039-1*} a membership test whose *tested*_simple_expression is a static expression, and whose membership_choice_list consists only of membership_choices that are either static *choice*_simple_expressions, static ranges, or subtype_marks that denote a static [(scalar or string)] subtype; 11/4

  Reason: Clearly, we should allow membership tests in exactly the same cases where we allow qualified_expressions. 11.a

- a short-circuit control form both of whose relations are static expressions; 12

- {*AI05-0147-1*} {*AI05-0188-1*} a conditional_expression all of whose conditions, *selecting*_expressions, and *dependent*_expressions are static expressions; 12.1/3

- {*AI12-0368-1*} a declare_expression whose *body*_expression is static and each of whose declarations, if any, is either the declaration of a static constant or is an object_renaming_declaration with an *object*_name that statically names the renamed object; 12.2/5

13 • a static expression enclosed in parentheses.

13.a **Discussion:** Informally, we talk about a *static value*. When we do, we mean a value specified by a static expression.

13.b/5 **Ramification:** {*AI12-0005-1*} The language requires a static expression in a number_declaration, a numeric type definition, certain representation items, and a number of other contexts.

14 A name *statically denotes* an entity if it denotes the entity and:

15 • It is a direct_name, expanded name, or character_literal, and it denotes a declaration other than a renaming_declaration; or

16 • It is an attribute_reference whose prefix statically denotes some entity; or

16.1/5 • {*AI12-0322-1*} It is a target_name (see 5.2.1) in an assignment_statement whose *variable_*name statically denotes some entity; or

17 • It denotes a renaming_declaration with a name that statically denotes the renamed entity.

17.a **Ramification:** Selected_components that are not expanded names and indexed_components do not statically denote things.

17.1/5 {*AI12-0368-1*} A name *statically names* an object if it:

17.2/5 • {*AI12-0368-1*} statically denotes the declaration of an object [(possibly through one or more renames)];

17.b/5 **Proof:** Follows from the definition of statically denotes.

17.3/5 • {*AI12-0368-1*} {*AI12-0373-1*} is a selected_component whose prefix statically names an object, there is no implicit dereference of the prefix, and the selector_name does not denote a component_declaration occurring within a variant_part; or

17.c/5 **Reason:** We disallow components in a variant_part so that no discriminant checks are needed to evaluate the selected_component. Note that other kinds of discriminant-dependent components do not need any checks on access (only when they are changed).

17.4/5 • {*AI12-0368-1*} is an indexed_component whose prefix statically names an object, there is no implicit dereference of the prefix, the object is statically constrained, and the index expressions of the object are static and have values that are within the range of the index constraint.

17.5/5 {*AI12-0368-1*} For an entity other than an object, a name statically names an entity if the name statically denotes the entity.

18 A *static function* is one of the following:

18.a **Ramification:** These are the functions whose calls can be static expressions.

19 • a predefined operator whose parameter and result types are all scalar types none of which are descendants of formal scalar types;

19.1/5 • {*AI12-0201-1*} a predefined relational operator whose parameters are of a string type that is not a descendant of a formal array type;

20/5 • {*AI12-0201-1*} a predefined concatenation operator whose result type is a string type that is not a descendant of a formal array type;

20.1/5 • {*AI12-0385-1*} a shifting or rotating function associated with a modular type declared in package Interfaces (see B.2);

21 • an enumeration literal;

21.1/5 • {*AI12-0075-1*} a static expression function (see 6.8);

22 • a language-defined attribute that is a function, if the prefix denotes a static scalar subtype, and if the parameter and result types are scalar.

23 In any case, a generic formal subprogram is not a static function.

{*AI12-0201-1*} {*AI12-0393-1*} A *static constant* is a constant view declared by a full constant declaration or an object_renaming_declaration with a static nominal subtype, having a value defined by a static scalar expression or by a static string expression, and which satisfies any constraint or predicate that applies to the nominal subtype. 24/5

Ramification: A deferred constant is not static; the view introduced by the corresponding full constant declaration can be static. 24.a

This paragraph was deleted.{*AI05-0229-1*} {*AI12-0201-1*} 24.b/5

This paragraph was deleted.{*AI12-0201-1*} 24.c/5

A *static range* is a range whose bounds are static expressions, [or a range_attribute_reference that is equivalent to such a range.] A *static discrete_range* is one that is a static range or is a subtype_indication that defines a static scalar subtype. The base range of a scalar type is a static range, unless the type is a descendant of a formal scalar type. 25

{*AI95-00263-01*} {*AI05-0153-3*} A *static subtype* is either a *static scalar subtype* or a *static string subtype*. A static scalar subtype is an unconstrained scalar subtype whose type is not a descendant of a formal type, or a constrained scalar subtype formed by imposing a compatible static constraint on a static scalar subtype. A static string subtype is an unconstrained string subtype whose index subtype and component subtype are static, or a constrained string subtype formed by imposing a compatible static constraint on a static string subtype. In any case, the subtype of a generic formal object of mode **in out**, and the result subtype of a generic formal function, are not static. Also, a subtype is not static if any Dynamic_Predicate specifications apply to it. 26/3

Ramification: String subtypes are the only composite subtypes that can be static. 26.a

Reason: The part about generic formal objects of mode **in out** is necessary because the subtype of the formal is not required to have anything to do with the subtype of the actual. For example: 26.b

```
subtype Int10 is Integer range 1..10;                                    26.c
generic                                                                  26.d
    F : in out Int10;
procedure G;
procedure G is                                                           26.e
begin
    case F is
        when 1..10 => null;
        -- Illegal!
    end case;
end G;
X : Integer range 1..20;                                                 26.f
procedure I is new G(F => X); -- OK.
```

The case_statement is illegal, because the subtype of F is not static, so the choices have to cover all values of Integer, not just those in the range 1..10. A similar issue arises for generic formal functions, now that function calls are object names. 26.g

The different kinds of *static constraint* are defined as follows: 27

- A null constraint is always static; 28
- A scalar constraint is static if it has no range_constraint, or one with a static range; 29
- An index constraint is static if each discrete_range is static, and each index subtype of the corresponding array type is static; 30
- A discriminant constraint is static if each expression of the constraint is static, and the subtype of each discriminant is static. 31

{*AI95-00311-01*} In any case, the constraint of the first subtype of a scalar formal type is neither static nor null. 31.1/2

32 A subtype is *statically constrained* if it is constrained, and its constraint is static. An object is *statically constrained* if its nominal subtype is statically constrained, or if it is a static string constant.

Legality Rules

32.1/3 {*AI05-0147-1*} An expression is *statically unevaluated* if it is part of:

32.2/3 - {*AI05-0147-1*} the right operand of a static short-circuit control form whose value is determined by its left operand; or

32.3/3 - {*AI05-0147-1*} {*AI05-0188-1*} a *dependent_*expression of an if_expression whose associated condition is static and equals False; or

32.4/3 - {*AI05-0147-1*} {*AI05-0188-1*} a condition or *dependent_*expression of an if_expression where the condition corresponding to at least one preceding *dependent_*expression of the if_expression is static and equals True; or

32.a/3 **Reason:** We need this bullet so that only a single *dependent_*expression is evaluated in a static if_expression if there is more than one condition that evaluates to True. The part about conditions makes

32.b/3
```
(if N = 0 then Min elsif 10_000/N > Min then 10_000/N else Min)
```

32.c/3 legal if N and Min are static and N = 0.

32.d/3 **Discussion:** {*AI05-0147-1*} {*AI05-0188-1*} We need the "of the if_expression" here so there is no confusion for nested if_expressions; this rule only applies to the conditions and *dependent_*expressions of a single if_expression. Similar reasoning applies to the "of a case_expression" of the last bullet.

32.5/3 - {*AI05-0188-1*} {*AI05-0269-1*} a *dependent_*expression of a case_expression whose *selecting_*expression is static and whose value is not covered by the corresponding discrete_choice_list; or

32.6/4 - {*AI05-0158-1*} {*AI12-0039-1*} a *choice_*simple_expression (or a simple_expression of a range that occurs as a membership_choice of a membership_choice_list) of a static membership test that is preceded in the enclosing membership_choice_list by another item whose individual membership test (see 4.5.2) statically yields True.

33/3 {*AI05-0147-1*} A static expression is evaluated at compile time except when it is statically unevaluated. The compile-time evaluation of a static expression is performed exactly, without performing Overflow_Checks. For a static expression that is evaluated:

34/3 - {*AI05-0262-1*} The expression is illegal if its evaluation fails a language-defined check other than Overflow_Check. For the purposes of this evaluation, the assertion policy is assumed to be Check.

34.a/3 **Reason:** {*AI05-0262-1*} Assertion policies can control whether checks are made, but we don't want assertion policies to affect legality. For Ada 2012, subtype predicates are the only checks controlled by the assertion policy that can appear in static expressions.

35/2 - {*AI95-00269-01*} If the expression is not part of a larger static expression and the expression is expected to be of a single specific type, then its value shall be within the base range of its expected type. Otherwise, the value may be arbitrarily large or small.

35.a/2 **Ramification:** {*AI95-00269-01*} If the expression is expected to be of a universal type, or of "any integer type", there are no limits on the value of the expression.

36/2 - {*AI95-00269-01*} If the expression is of type *universal_real* and its expected type is a decimal fixed point type, then its value shall be a multiple of the *small* of the decimal type. This restriction does not apply if the expected type is a descendant of a formal scalar type (or a corresponding actual type in an instance).

36.a **Ramification:** This means that a numeric_literal for a decimal type cannot have "extra" significant digits.

36.b/2 **Reason:** {*AI95-00269-01*} The small is not known for a generic formal type, so we have to exclude formal types from this check.

{*AI95-00269-01*} In addition to the places where Legality Rules normally apply (see 12.3), the above restrictions also apply in the private part of an instance of a generic unit. 37/2

Discussion: Values outside the base range are not permitted when crossing from the "static" domain to the "dynamic" domain. This rule is designed to enhance portability of programs containing static expressions. Note that this rule applies to the exact value, not the value after any rounding or truncation. (See below for the rounding and truncation requirements.) 37.a

Short-circuit control forms are a special case: 37.b

```
N: constant := 0.0;                                                    37.c
X: constant Boolean := (N = 0.0) or else (1.0/N > 0.5); -- Static.
```

The declaration of X is legal, since the divide-by-zero part of the expression is not evaluated. X is a static constant equal to True. 37.d

{*AI12-0075-1*} The preceding "statically unevaluated" rule allows 37.e/5

```
X : constant := (if True then 37 else (1 / 0));                        37.f/5
```

but does not allow 37.g/5

```
function If_Then_Else (Flag : Boolean; X, Y : Integer) return Integer is   37.h/5
   (if Flag then X else Y) with Static; -- see 6.8
X : constant := If_Then_Else (True, 37, 1 / 0);
```

because evaluation of a function call includes evaluation of all of its actual parameters. 37.i/5

Implementation Requirements

{*AI95-00268-01*} {*AI95-00269-01*} For a real static expression that is not part of a larger static expression, and whose expected type is not a descendant of a formal type, the implementation shall round or truncate the value (according to the Machine_Rounds attribute of the expected type) to the nearest machine number of the expected type; if the value is exactly half-way between two machine numbers, the rounding performed is implementation-defined. If the expected type is a descendant of a formal type, or if the static expression appears in the body of an instance of a generic unit and the corresponding expression is nonstatic in the corresponding generic body, then no special rounding or truncating is required — normal accuracy rules apply (see Annex G). 38/2

Implementation defined: Rounding of real static expressions which are exactly half-way between two machine numbers. 38.a.1/2

Reason: {*AI95-00268-01*} Discarding extended precision enhances portability by ensuring that the value of a static constant of a real type is always a machine number of the type. 38.a/2

When the expected type is a descendant of a formal floating point type, extended precision (beyond that of the machine numbers) can be retained when evaluating a static expression, to ease code sharing for generic instantiations. For similar reasons, normal (nondeterministic) rounding or truncating rules apply for descendants of a formal fixed point type. 38.b

{*AI95-00269-01*} There is no requirement for exact evaluation or special rounding in an instance body (unless the expression is static in the generic body). This eliminates a potential contract issue where the exact value of a static expression depends on the actual parameters (which could then affect the legality of other code). 38.b.1/2

Implementation Note: Note that the implementation of static expressions has to keep track of plus and minus zero for a type whose Signed_Zeros attribute is True. 38.c

{*AI95-00100-01*} Note that the only machine numbers of a fixed point type are the multiples of the small, so a static conversion to a fixed-point type, or division by an integer, must do truncation to a multiple of small. It is not correct for the implementation to do all static calculations in infinite precision. 38.d/2

Implementation Advice

{*AI95-00268-01*} For a real static expression that is not part of a larger static expression, and whose expected type is not a descendant of a formal type, the rounding should be the same as the default rounding for the target system. 38.1/2

Implementation Advice: A real static expression with a nonformal type that is not part of a larger static expression should be rounded the same as the target system. 38.e/2

NOTE 1 An expression can be static even if it occurs in a context where staticness is not required. 39

39.a **Ramification:** For example:

39.b
```
X : Float := Float'(1.0E+400) + 1.0 - Float'(1.0E+400);
```

39.c The expression is static, which means that the value of X must be exactly 1.0, independent of the accuracy or range of the run-time floating point implementation.

39.d The following kinds of expressions are never static: explicit_dereference, indexed_component, slice, **null**, aggregate, allocator.

40 NOTE 2 A static (or run-time) type_conversion from a real type to an integer type performs rounding. If the operand value is exactly half-way between two integers, the rounding is performed away from zero.

40.a **Reason:** We specify this for portability. The reason for not choosing round-to-nearest-even, for example, is that this method is easier to undo.

40.b **Ramification:** The attribute Truncation (see A.5.3) can be used to perform a (static) truncation prior to conversion, to prevent rounding.

40.c **Implementation Note:** The value of the literal 0E999999999999999999999999999999999999999999999999 is zero. The implementation must take care to evaluate such literals properly.

Examples

41 *Examples of static expressions:*

42
```
1 + 1        -- 2
abs(-10)*3   -- 30
```

43
```
Kilo : constant := 1000;
Mega : constant := Kilo*Kilo;    -- 1_000_000
Long : constant := Float'Digits*2;
```

44
```
Half_Pi    : constant := Pi/2;            -- see 3.3.2
Deg_To_Rad : constant := Half_Pi/90;
Rad_To_Deg : constant := 1.0/Deg_To_Rad;
                         -- equivalent to 1.0/((3.14159_26536/2)/90)
```

Extensions to Ada 83

44.a The rules for static expressions and static subtypes are generalized to allow more kinds of compile-time-known expressions to be used where compile-time-known values are required, as follows:

44.b - Membership tests and short-circuit control forms may appear in a static expression.

44.c - The bounds and length of statically constrained array objects or subtypes are static.

44.d - The Range attribute of a statically constrained array subtype or object gives a static range.

44.e - A type_conversion is static if the subtype_mark denotes a static scalar subtype and the operand is a static expression.

44.f - All numeric literals are now static, even if the expected type is a formal scalar type. This is useful in case_statements and variant_parts, which both now allow a value of a formal scalar type to control the selection, to ease conversion of a package into a generic package. Similarly, named array aggregates are also permitted for array types with an index type that is a formal scalar type.

44.g The rules for the evaluation of static expressions are revised to require exact evaluation at compile time, and force a machine number result when crossing from the static realm to the dynamic realm, to enhance portability and predictability. Exact evaluation is not required for descendants of a formal scalar type, to simplify generic code sharing and to avoid generic contract model problems.

44.h Static expressions are legal even if an intermediate in the expression goes outside the base range of the type. Therefore, the following will succeed in Ada 95, whereas it might raise an exception in Ada 83:

44.i
```
type Short_Int is range -32_768 .. 32_767;
I : Short_Int := -32_768;
```

44.j This might raise an exception in Ada 83 because "32_768" is out of range, even though "–32_768" is not. In Ada 95, this will always succeed.

44.k Certain expressions involving string operations (in particular concatenation and membership tests) are considered static in Ada 95.

44.l The reason for this change is to simplify the rule requiring compile-time-known string expressions as the link name in an interfacing pragma, and to simplify the preelaborability rules.

Incompatibilities With Ada 83

An Ada 83 program that uses an out-of-range static value is illegal in Ada 95, unless the expression is part of a larger static expression, or the expression is not evaluated due to being on the right-hand side of a short-circuit control form. 44.m

Wording Changes from Ada 83

{*AI05-0299-1*} This subclause (and 4.5.5, “Multiplying Operators”) subsumes the RM83 section on Universal Expressions. 44.n/3

The existence of static string expressions necessitated changing the definition of static subtype to include string subtypes. Most occurrences of "static subtype" have been changed to "static scalar subtype", in order to preserve the effect of the Ada 83 rules. This has the added benefit of clarifying the difference between "static subtype" and "statically constrained subtype", which has been a source of confusion. In cases where we allow static string subtypes, we explicitly use phrases like "static string subtype" or "static (scalar or string) subtype", in order to clarify the meaning for those who have gotten used to the Ada 83 terminology. 44.o

In Ada 83, an expression was considered nonstatic if it raised an exception. Thus, for example: 44.p

```
Bad: constant := 1/0; -- Illegal!
```
44.q

was illegal because 1/0 was not static. In Ada 95, the above example is still illegal, but for a different reason: 1/0 is static, but there's a separate rule forbidding the exception raising. 44.r

Inconsistencies With Ada 95

{*AI95-00268-01*} **Amendment Correction:** Rounding of static real expressions is implementation-defined in Ada 2005, while it was specified as away from zero in (original) Ada 95. This could make subtle differences in programs. However, the original Ada 95 rule required rounding that (probably) differed from the target processor, thus creating anomalies where the value of a static expression was required to be different than the same expression evaluated at run time. 44.s/2

Wording Changes from Ada 95

{*AI95-00263-01*} {*AI95-00268-01*} The Ada 95 wording that defined static subtypes unintentionally failed to exclude formal derived types that happen to be scalar (these aren't formal scalar types); and had a parenthetical remark excluding formal string types - but that was neither necessary nor parenthetical (it didn't follow from other wording). This issue also applies to the rounding rules for real static expressions. 44.t/2

{*AI95-00269-01*} Ada 95 didn't clearly define the bounds of a value of a static expression for universal types and for “any integer/float/fixed type”. We also make it clear that we do not intend exact evaluation of static expressions in an instance body if the expressions aren't static in the generic body. 44.u/2

{*AI95-00311-01*} We clarify that the first subtype of a scalar formal type has a nonstatic, nonnull constraint. 44.v/2

Wording Changes from Ada 2005

{*AI05-0147-1*} {*AI05-0188-1*} Added wording to define staticness and the lack of evaluation for if_expressions and case_expressions. These are new and defined elsewhere. 44.w/3

{*AI05-0153-3*} Added wording to prevent subtypes that have dynamic predicates (see 3.2.4) from being static. 44.x/3

{*AI05-0158-1*} Revised wording for membership tests to allow for the new possibilities allowed by the membership_choice_list. 44.y/3

Incompatibilities With Ada 2012

{*AI12-0201-1*} Added a missing exclusion for concatenations of a string type descended from a formal array type. This could potentially make some expression non-static; but as that could only matter in a context where a static string is required (such as the Link_Name aspect), it is quite unlikely. 44.z/5

{*AI12-0385-1*} Shifting and rotating functions declared in package Interfaces are now static. This could potentially make some expression illegal that is legal if nonstatic (as in Ada 2012). While this can happen especially in conditional code that is not in use, it is quite unlikely given typical uses of shifting or rotating functions. 44.aa/5

Extensions to Ada 2012

{*AI12-0201-1*} Expressions involving string relational operators or string type conversions now can be static. Additionally, the length limit on static string constants was removed as being a hazard without much help to implementations. 44.bb/5

Wording Changes from Ada 2012

44.cc/5 {*AI12-0064-2*} Defined the staticness of the Nonblocking attribute (see 9.5).

44.dd/5 {*AI12-0075-1*} Expression functions can be static if declared correctly; this is documented as an extension in 6.8.

44.ee/5 {*AI12-0249-1*} {*AI12-0394-1*} A numeric_literal or named number can be non-static if they interpreted using an Integer_Literal or Real_Literal aspect (see 4.2.1).

44.ff/5 {*AI12-0322-1*} Clarified that a target name symbol can statically denote an entity if the associated *variable*_name statically denotes an entity. This is necessary so that target names participate in the anti-order-dependence checks of 6.4.1.

44.gg/3 {*AI12-0368-1*} {*AI12-0373-1*} Added wording to define staticness for declare_expressions. Also moved "statically names" definition here and used it in array attribute prefix wording.

44.hh/5 {*AI12-0393-1*} **Correction:** Clarified that constants whose values do not belong to their nominal subtype are not static. This change potentially would be incompatible, but this case is considered pathological and will not be checked by the ACATS.

4.9.1 Statically Matching Constraints and Subtypes

Static Semantics

1/2 {*AI95-00311-01*} A constraint *statically matches* another constraint if:

1.1/2 - both are null constraints;

1.a/5 **Discussion:** A null constraint has nothing to do with null exclusions! Unconstrained array subtypes, subtypes with unknown discriminants, and subtypes with no explicit constraint have null constraints (see 3.2). This terminology became confusing when null exclusions were introduced in the 2007 Amendment.

1.2/2 - both are static and have equal corresponding bounds or discriminant values;

1.3/2 - both are nonstatic and result from the same elaboration of a constraint of a subtype_indication or the same evaluation of a range of a discrete_subtype_definition; or

1.4/2 - {*AI95-00311-01*} both are nonstatic and come from the same formal_type_declaration.

1.5/5 {*AI12-0374-2*} {*AI12-0427-1*} The Global or Global'Class aspects (see 6.1.2) of two entities *statically match* if both consist of a single global_aspect_definition where each is the reserved word **null**, or each is of the form "global_mode global_designator" with each global_mode being the same sequence of reserved words and each global_designator being the same reserved word, or each being a global_name that statically names the same entity.

2/5 {*AI95-00231-01*} {*AI95-00254-01*} {*AI05-0153-3*} {*AI12-0059-1*} {*AI12-0374-2*} A subtype *statically matches* another subtype of the same type if they have statically matching constraints, all predicate specifications that apply to them come from the same declarations, Nonblocking aspects have the same value, global aspects statically match, Object_Size (see 13.3) has been specified to have a nonconfirming value for either both or neither, and the nonconfirming values, if any, are the same, and, for access subtypes, either both or neither exclude null. Two anonymous access-to-object subtypes statically match if their designated subtypes statically match, and either both or neither exclude null, and either both or neither are access-to-constant. Two anonymous access-to-subprogram subtypes statically match if their designated profiles are subtype conformant, and either both or neither exclude null.

2.a **Ramification:** Statically matching constraints and subtypes are the basis for subtype conformance of profiles (see 6.3.1).

2.b/2 **Reason:** Even though anonymous access types always represent different types, they can statically match. That's important so that they can be used widely. For instance, if this wasn't true, access parameters and access discriminants could never conform, so they couldn't be used in separate specifications.

2.c/5 **Ramification:** {*AI12-0059-1*} If one of the subtypes is not yet frozen, an implementation may have to repeat the check when the subtypes are both frozen (as it is impossible to check the Object_Size part before the subtypes are frozen). This recheck can only make a previously statically matching subtype fail to match; it cannot make a match legal.

Discussion: {*AI12-0059-1*} We exclude the case where both Object_Sizes are confirming so that we don't introduce an incompatibility for existing Ada code. But practically the implementation can simply check that the Object_Size values are the same, as we have a rule in 13.1 that the subtype-specific aspects (such as Object_Size) are always the same for statically matching subtypes. We wrote the rules this way to avoid having wording that appeared to require predicting the future ("would statically match if ...").

2.d/5

Two ranges of the same type *statically match* if both result from the same evaluation of a range, or if both are static and have equal corresponding bounds.

3

Ramification: The notion of static matching of ranges is used in 12.5.3, “Formal Array Types”; the index ranges of formal and actual constrained array subtypes have to statically match.

3.a

{*AI05-0086-1*} {*AI05-0153-3*} A constraint is *statically compatible* with a scalar subtype if it statically matches the constraint of the subtype, or if both are static and the constraint is compatible with the subtype. A constraint is *statically compatible* with an access or composite subtype if it statically matches the constraint of the subtype, or if the subtype is unconstrained.

4/3

Discussion: Static compatibility is required when constraining a parent subtype with a discriminant from a new discriminant_part. See 3.7. Static compatibility is also used in matching generic formal derived types.

4.a

Note that statically compatible with a subtype does not imply compatible with a type. It is OK since the terms are used in different contexts.

4.b

{*AI05-0153-3*} Two statically matching subtypes are statically compatible with each other. In addition, a subtype *S1* is statically compatible with a subtype *S2* if:

5/3

- the constraint of *S1* is statically compatible with *S2*, and 6/3
- {*AI05-0086-1*} if *S2* excludes null, so does *S1*, and 7/3
- either: 8/3
  - all predicate specifications that apply to *S2* apply also to *S1*, or 9/3
  - {*AI05-0290-1*} {*AI12-0071-1*} both subtypes are static, every value that satisfies the predicates of *S1* also satisfies the predicates of *S2*, and it is not the case that both types each have at least one applicable predicate specification, predicate checks are enabled (see 11.4.2) for *S2*, and predicate checks are not enabled for *S1*. 10/4

Wording Changes from Ada 83

This subclause is new to Ada 95.

10.a

Wording Changes from Ada 95

{*AI95-00231-01*} {*AI95-00254-01*} Added static matching rules for null exclusions and anonymous access-to-subprogram types; both of these are new.

10.b/2

{*AI95-00311-01*} We clarify that the constraint of the first subtype of a scalar formal type statically matches itself.

10.c/2

Incompatibilities With Ada 2005

{*AI05-0086-1*} **Correction:** Updated the statically compatible rules to take null exclusions into account. This is technically incompatible, as it could cause a legal Ada 2005 program to be rejected; however, such a program violates the intent of the rules (for instance, 3.7(15)) and this probably will simply detect bugs.

10.d/3

Wording Changes from Ada 2005

{*AI05-0153-3*} {*AI05-0290-1*} Modified static matching and static compatibility to take predicate aspects (see 3.2.4) into account.

10.e/3

Wording Changes from Ada 2012

{*AI12-0071-1*} **Corrigendum:** Updated wording of static compatibility to use the new term "satisfies the predicates" (see 3.2.4).

10.f/4

{*AI12-0059-1*} Updated wording to take nonconfirming values of Object_Size into account.

10.g/5

{*AI12-0374-2*} Static matching now includes the effects of Global aspects and the Nonblocking aspect.

10.h/5

4.10 Image Attributes

1/5 {*AI12-0315-1*} An *image* of a value is a string representing the value in display form. The attributes Image, Wide_Image, and Wide_Wide_Image are available to produce the image of a value as a String, Wide_String, or Wide_Wide_String (respectively). User-defined images for a given type can be implemented by overriding the default implementation of the attribute Put_Image.

Static Semantics

2/5 {*AI12-0020-1*} {*AI12-0315-1*} For every subtype S of a type T other than *universal_real* or *universal_fixed*, the following type-related operational attribute is defined:

3/5 S'Put_Image

{*AI12-0020-1*} {*AI12-0320-1*} {*AI12-0340-1*} S'Put_Image denotes a procedure with the following specification:

3.1/5
```
procedure S'Put_Image
   (Buffer : in out
             Ada.Strings.Text_Buffers.Root_Buffer_Type'Class;
    Arg    : in T);
```

3.2/5 {*AI12-0315-1*} {*AI12-0340-1*} The default implementation of S'Put_Image writes (using Wide_Wide_Put) an *image* of the value of *Arg*.

4/5 {*AI12-0020-1*} {*AI12-0419-1*} The Put_Image attribute may be specified for any specific type T either via an attribute_definition_clause or via an aspect_specification specifying the Put_Image aspect of the type.[The Put_Image aspect is not inherited, but rather is implicitly composed for derived types, as defined below.]

4.a/5 **Aspect Description for Put_Image:** Procedure to define the image of a given type.

4.b/5 **Discussion:** In contrast, the Image, Wide_Image, and Wide_Wide_Image attributes and their associated aspects can not be specified. The behavior of any of these attributes is defined in terms of calls to the corresponding Put_Image procedure, so changes in their behavior may be accomplished via a Put_Image specification.

4.c/5 In earlier versions of Ada, Image and related attributes were defined only for scalar types. The definition of these attributes is now very different, but there should be no change in the behavior of existing programs as a result of these changes.

5/5 {*AI12-0427-1*} For an aspect_specification or attribute_definition_clause specifying Put_Image, the subtype of the *Arg* parameter shall be the first subtype or the base subtype if scalar, and the first subtype if not scalar.

6/5 {*AI12-0020-1*} {*AI12-0419-1*} The behavior of the default implementation of S'Put_Image depends on the class of T.

7/5 {*AI12-0020-1*} {*AI12-0419-1*} {*AI12-0435-1*} For an untagged derived type, or a null extension, the default implementation of T'Put_Image invokes the Put_Image for its parent type on a conversion of the parameter of type T to the parent type.

8/5 {*AI12-0020-1*} {*AI12-0340-1*} {*AI12-0384-2*} {*AI12-0419-1*} For a nonderived elementary type, the implementation is equivalent to:

9/5
```
procedure Scalar_Type'Put_Image
  (Buffer : in out Ada.Strings.Text_Buffers.Root_Buffer_Type'Class;
   Arg    : in Scalar_Type) is
begin
   Buffer.Wide_Wide_Put (<described below>);
end Scalar_Type'Put_Image;
```

10/5 where the Wide_Wide_String value written out to the text buffer is defined as follows:

- {*AI12-0020-1*} For an integer type, the image written out is the corresponding decimal literal, without underlines, leading zeros, exponent, or trailing spaces, but with a single leading character that is either a minus sign or a space. 11/5

  Implementation Note: If the machine supports negative zeros for signed integer types, it is not specified whether " 0" or "-0" should be returned for negative zero. We don't have enough experience with such machines to know what is appropriate, and what other languages do. In any case, the implementation should be consistent. 11.a/5

  Discussion: We allow S'Put_Image when S is *universal_integer* or *root_integer*, because the details of the desired string do not depend on properties of an integer type. While S'Put_Image cannot be called directly for these types (as they cannot be named), it can be called as part of evaluating an Image attribute. Note that other rules of the language ensure that an implementation can evaluate any *universal_integer* attribute using type *root_integer*; therefore, Constraint_Error could be raised by the evaluation of an Image attribute if the value of the prefix is outside of the range of *root_integer*. 11.b/5

- {*AI12-0020-1*} For an enumeration type, the image written out is either the corresponding identifier in upper case or the corresponding character literal (including the two apostrophes); neither leading nor trailing spaces are included. For a *nongraphic character* (a value of a character type that has no enumeration literal associated with it), the value is a corresponding language-defined name in upper case (for example, the image of the nongraphic character identified as *nul* is `"NUL"` — the quotes are not part of the image). 12/5

  Implementation Note: For an enumeration type T that has "holes" (caused by an enumeration_representation_clause), T'Put_Image should raise Program_Error if the value is one of the holes (which is a bounded error anyway, since holes can be generated only via uninitialized variables and similar things). 12.a/5

- {*AI12-0020-1*} For a floating point type, the image written out is a decimal real literal best approximating the value (rounded away from zero if halfway between) with a single leading character that is either a minus sign or a space, a single digit (that is nonzero unless the value is zero), a decimal point, S'Digits-1 (see 3.5.8) digits after the decimal point (but one if S'Digits is one), an upper case E, the sign of the exponent (either + or -), and two or more digits (with leading zeros if necessary) representing the exponent. If S'Signed_Zeros is True, then the leading character is a minus sign for a negatively signed zero. 13/5

  To be honest: Leading zeros are present in the exponent only if necessary to make the exponent at least two digits. 13.a/5

  Reason: This image is intended to conform to that produced by Text_IO.Float_IO.Put in its default format. 13.b/5

  Implementation Note: The rounding direction is specified here to ensure portability of output results. 13.c/5

  Reason: {*AI12-0315-1*} We do not allow S'Put_Image when S is *universal_real*, as the details of the desired string depend on the properties of the (specific) type of S. Specifically, *universal_real* does not have a defined value for S'Digits. 13.d/5

- {*AI12-0020-1*} For a fixed point type, the image written out is a decimal real literal best approximating the value (rounded away from zero if halfway between) with a single leading character that is either a minus sign or a space, one or more digits before the decimal point (with no redundant leading zeros), a decimal point, and S'Aft (see 3.5.10) digits after the decimal point. 14/5

  Reason: This image is intended to conform to that produced by Text_IO.Fixed_IO.Put. 14.a/5

  Implementation Note: The rounding direction is specified here to ensure portability of output results. 14.b/5

  For a machine that supports negative zeros, it is not specified whether " 0.000" or "-0.000" is returned. See corresponding comment above about integer types with signed zeros. 14.c/5

  Reason: {*AI12-0315-1*} We do not allow S'Put_Image when S is *universal_fixed*, as the details of the desired string depend on the properties of the (specific) type of S. Specifically, *universal_fixed* does not have a defined value for S'Aft. 14.d/5

- {*AI12-0020-1*} For an access type (named or anonymous), the image written out depends on whether the value is **null**. If it is **null**, then the image is `"NULL"`. Otherwise the image is a left parenthesis followed by `"ACCESS"`, a space, and a sequence of graphic characters, other than space or right parenthesis, representing the location of the designated object, followed by a right parenthesis, as in `"(ACCESS FF0012AC)"`. 15/5

  To be honest: {*AI12-0315-1*} S'Put_Image is defined for *universal_access*, but it can never be called (as no legal prefix of Image has that type, and that type cannot be named preventing direct calls). 15.a/5

16/5 {*AI12-0020-1*} {*AI12-0419-1*} {*AI12-0435-1*} For a nonnull type extension, the default implementation of T'Put_Image depends on whether there exists a noninterface ancestor of T (other than T itself) for which the Put_Image aspect has been directly specified. If so, then T'Put_Image will generate an image based on extension aggregate syntax where the ancestor type of the extension aggregate is the nearest ancestor type whose Put_Image aspect has been specified. If no such ancestor exists, then the default implementation of T'Put_Image is the same as described below for a nonderived record type.

16.a/5 **Discussion:** This might generate an image such as `"(This Text Was User-Generated with C1 => 123, C2 =>  456)"` where the "This Text was User-Generated" portion of the text was generated by the call to the user-specified Put_Image routine.

17/5 {*AI12-0419-1*} For a specific, nonderived composite type:

18/5 - {*AI12-0020-1*} If the default implementation of Put_Image writes components, the order in which components are written is the same canonical order in which components of a composite type T are written out by the default implementation of T'Write. [This is also the order that is used in determining the meaning of a positional aggregate of type T.]

18.a/5 **Discussion:** {*AI12-0020-1*} {*AI12-0384-2*} In general, the default implementation of T'Put_Image for a composite type will involve some sequence of calls to Put and its Wide and Wide_Wide variants and calls to the Put_Image procedures of component types and, in the case of an array type, index types. The Put calls may pass in either literal values (e.g., "(", ")", "'(", " => ", or ", "), or other things (such as component names for record values, task_id images for tasks, or the Wide_Wide_Expanded_Name of the tag in the class-wide case).

19/5 - {*AI12-0020-1*} {*AI12-0419-1*} For an array type T, the default implementation of T'Put_Image generates an image based on named (not positional) array aggregate syntax (with '[' and ']' as the delimiters) using calls to the Put_Image procedures of the index type(s) and the element type to generate images for values of those types.

19.a/5 **Discussion:** An array type might generate an image such as:

19.b/5
```
[ 1 => [ 1 => [ 123 => True,  124 => False]
         2 => [ 123 => False,  124 => False]],
  2 => [ 1 => [ 123 => True,  124 => True],
         2 => [ 123 => True,  124 => False]]]
```

19.c/5 although perhaps with different white space and/or line breaking.

20/5 {*AI12-0020-1*} The case of a null array is handled specially, using ranges for index bounds and `"<>"` as a syntactic component-value placeholder.

20.a/5 **Discussion:** This might generate an image such as `"[ 1 ..  3 => [ 1 ..  0 => [ 1 .. 5 => <>]]]"`, where the use of "<>" (among other things) indicates that the overall array is a null array and has no actual elements.

21/5 - {*AI12-0020-1*} {*AI12-0419-1*} For a record type (or, as indicated above, a type extension with no noninterface ancestor specifying Put_Image), or a protected type, the default implementation of T'Put_Image generates an image based on named (not positional) record aggregate syntax (except that for a protected type, the initial left parenthesis is followed by `"PROTECTED with "`). Component names are displayed in upper case, following the rules for the image of an enumeration value. Component values are displayed via calls to the component type's Put_Image procedure.

21.a/5 **Discussion:** This might generate an image such as:

21.b/5
```
"(FOO => [1 => 'c',  2 => 'a',  3 => 't'], BAR => TRUE)"
```

22/5 {*AI12-0020-1*} The image written out for a record having no components (including any interface type) is `"(NULL RECORD)"`. The image written out for a componentless protected type is `"(PROTECTED NULL RECORD)"`. In the case of a protected type T, a call to the default implementation of T'Put_Image begins only one protected (read-only) action.

22.a/5 **Implementation Note:** The expected, but not required, implementation model for generating the image of a protected record involves the compiler producing a "helper" protected function which T'Put_Image would call. The result type of this function might be a null record; it is only a function because it does not need a write-lock, not because it returns a meaningful result.

22.b/5 The assertion in the following example should succeed:

```
type T1 (D1, D2 : Positive) is record ... end record; -- untagged
type T2 (D : Positive) is new T1 (D1 => D, D2 => D);
X : T2 (D => 123) := ... ;
pragma Assert (X'Image /= T1(X)'Image);
```

22.c/5

- {*AI12-0020-1*} For an undiscriminated task type, the default implementation of T'Put_Image generates an image of the form "(TASK <task_id_image>)" where <task_id_image> is the result obtained by calling Task_Identification.Image with the id of the given task and then passing that String to Characters.Conversions.To_Wide_Wide_String. 23/5

- {*AI12-0020-1*} For a discriminated task type, the default implementation of T'Put_Image also includes discriminant values, as in: 24/5

  "(TASK <task_id_image> with D1 => 123, D2 => 456)" 25/5

{*AI12-0020-1*} {*AI12-0384-2*} For a class-wide type, the default implementation of T'Put_Image generates an image based on qualified expression syntax. Wide_Wide_Put is called with Wide_Wide_Expanded_Name of *Arg*'Tag. Then S'Put_Image is called, where S is the specific type identified by *Arg*'Tag. 26/5

Implementation Note: This will typically require a dispatching call. 26.a/5

Discussion: This might generate an image such as: 26.b/5

```
SOME_PACKAGE.SOME_TAGGED_TYPE'
   (COMPONENT_1 =>  123, COMPONENT_2 => 456)
```

26.c/5

The parentheses in this case are generated by the call to Some_Tagged_Type'Put_Image. 26.d/5

{*AI12-0020-1*} {*AI12-0384-2*} [T'Put_Image is the same for both the partial view and full view of T, if T has a partial view.] 27/5

Proof: {*AI12-0384-2*} A type-related operational aspect is the same for the full view and partial view of a type. See 13.1. 27.a/5

{*AI12-0435-1*} In the parameter_and_result_profile for the default implementation of Put_Image, the subtype of the *Arg* parameter is the base subtype of *T* if *T* is a scalar type, and the first subtype otherwise. For an aspect_specification or attribute_definition_clause specifying Put_Image, the subprogram name shall denote a nonabstract procedure whose second parameter is either of the first subtype of *T*, or as an option when *T* is scalar, the base subtype of *T*. 28/5

Ramification: Put_Image is never an abstract routine, even for an abstract type. Thus, Put_Image and Image can be called for any type. 28.a/5

{*AI12-0020-1*} For every subtype S of a type T, the following attributes are defined: 29/5

S'Wide_Wide_Image 30/5

{*AI12-0020-1*} {*AI12-0340-1*} {*AI12-0384-2*} S'Wide_Wide_Image denotes a function with the following specification:

```
function S'Wide_Wide_Image(Arg : S'Base)
  return Wide_Wide_String
```

30.1/5

S'Wide_Wide_Image calls S'Put_Image passing *Arg* (which will typically store a sequence of character values in a text buffer) and then returns the result of retrieving the contents of that buffer with function Wide_Wide_Get. The lower bound of the result is one. [Any exception propagated by the call of S'Put_Image is propagated.] 30.2/5

S'Wide_Image 31/5

{*AI12-0020-1*} {*AI12-0340-1*} {*AI12-0384-2*} S'Wide_Image denotes a function with the following specification:

```
function S'Wide_Image(Arg : S'Base)
  return Wide_String
```

32/5

S'Wide_Image calls S'Put_Image passing *Arg* (which will typically store a sequence of character values in a text buffer) and then returns the result of retrieving the contents of that 33/5

buffer with function Wide_Get. The lower bound of the result is one. [Any exception propagated by the call of S'Put_Image is propagated.]

34/5 S'Image {*AI12-0020-1*} {*AI12-0340-1*} {*AI12-0384-2*} S'Image denotes a function with the following specification:

35/5
```
function S'Image(Arg : S'Base)
  return String
```

36/5 S'Image calls S'Put_Image passing *Arg* (which will typically store a sequence of character values in a text buffer) and then returns the result of retrieving the contents of that buffer with function Get. The lower bound of the result is one. [Any exception propagated by the call of S'Put_Image is propagated.]

37/5 {*AI12-0124-1*} {*AI12-0020-1*} {*AI12-0315-1*} For a prefix X of a type T other than *universal_real* or *universal_fixed*, the following attributes are defined:

38/5 X'Wide_Wide_Image

{*AI12-0124-1*} {*AI12-0020-1*} X'Wide_Wide_Image denotes the result of calling function S'Wide_Wide_Image with *Arg* being X, where S is the nominal subtype of X.

39/5 X'Wide_Image

{*AI12-0124-1*} {*AI12-0020-1*} X'Wide_Image denotes the result of calling function S'Wide_Image with *Arg* being X, where S is the nominal subtype of X.

40/5 X'Image {*AI12-0124-1*} {*AI12-0020-1*} X'Image denotes the result of calling function S'Image with *Arg* being X, where S is the nominal subtype of X.

Implementation Permissions

41/5 {*AI12-0020-1*} An implementation may transform the image generated by the default implementation of S'Put_Image for a composite subtype S in the following ways:

42/5 • If S is a composite subtype, the leading character of the image *M* of a component value or index value is a space, and the immediately preceding character (if any) is an open parenthesis, open bracket, or space, then the leading space of the image *M* may be omitted.

42.a/5 **Discussion:** This means that it is permitted to generate `"[1 => 123, 2 => 456]"` instead of `"[ 1 =>  123, 2 =>  456]"`.

43/5 • If S is an array subtype, the low bound of the array in each dimension equals the low bound of the corresponding index subtype, and the array value is not a null array value, then positional array aggregate syntax may be used.

43.a/5 **Discussion:** This means that it is permitted to generate `"[TRUE, TRUE, FALSE]"` instead of `"[ 1 => TRUE, 2 => TRUE,  3 => FALSE]"` if the low bound of the index subtype is one.

44/5 • If S is an array subtype and the given value can be displayed using named_array_aggregate syntax where some discrete_choice_list identifies more than one index value by identifying a sequence of one or more ranges and values separated by vertical bars, then this image may be generated instead; this may involve the reordering of component values.

44.a/5 **Discussion:** This means that it is permitted to generate `"[ 1 ..  2 |  5 => TRUE,  3 ..  4 => FALSE]"` instead of `"[ 1 => TRUE,  2 => TRUE,  3 => FALSE,  4 => FALSE,  5 => TRUE]"`.

45/5 • Similarly, if S is a record subtype (or a discriminated type) and the given value can be displayed using named component association syntax where the length of some component_choice_list is greater than one, then this image may be generated instead; this may involve the reordering of component values.

45.a/5 **Discussion:** This means that it is permitted to generate `"(F1 | F2 => TRUE)"` instead of `"(F1 => TRUE, F2 => TRUE)"`.

46/5 • {*AI12-0020-1*} {*AI12-0340-1*} {*AI12-0384-2*} Additional spaces (Wide_Wide_Characters with position 32), and calls to the New_Line operation of a text buffer, may be inserted to improve

readability of the generated image, with the spaces inserted directly or via use of the Increase_Indent and Decrease_Indent procedures.

- {*AI12-0384-2*} For a string type, implementations may produce an image corresponding to a string literal. 47/5
- {*AI12-0384-2*} For an unchecked union type, implementations may raise Program_Error or produce some recognizable image (such as "`(UNCHECKED UNION)`") that does not require reading the discriminants. 48/5

{*AI12-0304-1*} For each language-defined nonscalar type T, T'Put_Image may be specified. 49/5

Discussion: This permission applies, in particular, to nonscalar types declared in language-defined generic packages, and to any language-defined private type, even if an implementation chooses to complete it as a scalar type. 49.a/5

Ramification: For any language-defined scalar type T, T'Put_Image should not be specified; the Image attribute needs to return the language-defined image for such types. This is important for compatibility: the Image attribute has been available for scalar types for many Ada revisions, and programs can (and do!) depend on its behavior. 49.b/5

Implementation Requirements

{*AI12-0304-1*} For each language-defined container type T (that is, each of the Vector, List, Map, Set, Tree, and Holder types defined in the various children of Ada.Containers), T'Put_Image shall be specified so that T'Image produces a result consistent with array aggregate syntax (using '[' and ']' as delimiters) as follows: 50/5

- Vector images shall be consistent with the default image of an array type with the same index and component types. 51/5

  Discussion: In particular, this means that the format is that of a named array aggregate. We have no recommendation on how to handle empty elements; if the implementation can identify them, it may wish to display them specially, but otherwise, they're just uninitialized elements. 51.a/5

- Map images shall be consistent with named array aggregate syntax, using key value images in place of discrete choice names. For example, `[Key1 => Value1, Key2 => Value2]`. 52/5

  Discussion: There is no recommendation about the order in which key/element pairs occur within a map image. In the case of multiple key values whose corresponding element values have the same image, there is no recommendation about factoring (that is, generating `Key1 | Key2 => Some_Value` instead of `Key1 => Some_Value, Key2 => Some_Value`). 52.a/5

- Set, List, and Holder images shall be consistent with positional array aggregate syntax. List elements shall occur in order within an image of a list. The image of an empty holder shall be `[]`. 53/5

  Discussion: There is no recommendation about the order in which set elements occur within the image of a set. 53.a/5

- Tree images (and images of subtrees of trees) shall be consistent with positional array aggregate syntax. For example, `[[1, 2], [111, 222, 333]]`. 54/5

{*AI12-0304-1*} For each language-defined nonscalar type T that has a primitive language-defined Image function whose profile is type conformant with that of T'Image (for example, Ada.Numerics.Float_Random.State has such an Image function), T'Put_Image shall be specified so that T'Image yields the same result as that Image function. 55/5

Implementation Advice

For each language-defined private type T, T'Image should generate an image that would be meaningful based only on the relevant public interfaces, as opposed to requiring knowledge of the implementation of the private type. 56/5

Implementation Advice: For each language-defined private type T, T'Image should generate an image that would be meaningful based only on the relevant public interfaces. 56.a/5

Extensions to Ada 2012

56.b/5 {*AI12-0020-1*} {*AI12-0315-1*} {*AI12-0340-1*} {*AI12-0384-2*} {*AI12-0419-1*} {*AI12-0427-1*} {*AI12-0435-1*} Attribute Put_Image is new. Attributes Image, Wide_Image, and Wide_Wide_Image now can be used with any type, and are defined in terms of Put_Image so that they can be redefined.

5 Statements

[A statement defines an action to be performed upon its execution.] 1

{*AI95-00318-02*} {*AI05-0299-1*} {*AI12-0449-1*} [This clause describes the general rules applicable to all statements. Some statements are discussed in later clauses: Procedure_call_statements and return statements are described in Clause 6, "Subprograms". Entry_call_statements, requeue_statements, delay_statements, accept_statements, select_statements, and abort_statements are described in Clause 9, "Tasks and Synchronization". Raise_statements are described in Clause 11, "Exceptions", and code_statements in Clause 13, "Representation Issues". The remaining forms of statements are presented in this clause.] 2/5

Wording Changes from Ada 83

{*AI95-00318-02*} The description of return statements has been moved to 6.5, "Return Statements", so that it is closer to the description of subprograms. 2.a/2

5.1 Simple and Compound Statements - Sequences of Statements

{*AI12-0119-1*} [A statement is either simple or compound. A simple_statement encloses no other statement. A compound_statement can enclose simple_statements and other compound_statements.] A *parallel construct* is a construct that introduces additional logical threads of control (see Clause 9) without creating a new task. Parallel loops (see 5.5) and parallel_block_statements (see 5.6.1) are parallel constructs. 1/5

Term entry: parallel construct — executable construct that defines multiple activities of a single task that can proceed in parallel, via the execution of multiple logical threads of control 1.a/5

Syntax

{*AI05-0179-1*} sequence_of_statements ::= statement {statement} {label} 2/3

statement ::= 3
  {label} simple_statement | {label} compound_statement

{*AI95-00318-02*} simple_statement ::= null_statement 4/2
  | assignment_statement | exit_statement
  | goto_statement | procedure_call_statement
  | simple_return_statement | entry_call_statement
  | requeue_statement | delay_statement
  | abort_statement | raise_statement
  | code_statement

{*AI95-00318-02*} {*AI12-0119-1*} compound_statement ::= 5/5
    if_statement | case_statement
  | loop_statement | block_statement
  | extended_return_statement
  | parallel_block_statement
  | accept_statement | select_statement

null_statement ::= **null**; 6

label ::= <<*label_*statement_identifier>> 7

statement_identifier ::= direct_name 8

The direct_name of a statement_identifier shall be an identifier (not an operator_symbol). 9

Name Resolution Rules

10 The direct_name of a statement_identifier shall resolve to denote its corresponding implicit declaration (see below).

Legality Rules

11 Distinct identifiers shall be used for all statement_identifiers that appear in the same body, including inner block_statements but excluding inner program units.

Static Semantics

12 For each statement_identifier, there is an implicit declaration (with the specified identifier) at the end of the declarative_part of the innermost block_statement or body that encloses the statement_identifier. The implicit declarations occur in the same order as the statement_identifiers occur in the source text. If a usage name denotes such an implicit declaration, the entity it denotes is the label, loop_statement, or block_statement with the given statement_identifier.

12.a **Reason:** We talk in terms of individual statement_identifiers here rather than in terms of the corresponding statements, since a given statement may have multiple statement_identifiers.

12.b A block_statement that has no explicit declarative_part has an implicit empty declarative_part, so this rule can safely refer to the declarative_part of a block_statement.

12.c The scope of a declaration starts at the place of the declaration itself (see 8.2). In the case of a label, loop, or block name, it follows from this rule that the scope of the implicit declaration starts before the first explicit occurrence of the corresponding name, since this occurrence is either in a statement label, a loop_statement, a block_statement, or a goto_statement. An implicit declaration in a block_statement may hide a declaration given in an outer program unit or block_statement (according to the usual rules of hiding explained in 8.3).

12.d The syntax rule for label uses statement_identifier which is a direct_name (not a defining_identifier), because labels are implicitly declared. The same applies to loop and block names. In other words, the label itself is not the defining occurrence; the implicit declaration is.

12.e We cannot consider the label to be a defining occurrence. An example that can tell the difference is this:

12.f
```
declare
    -- Label Foo is implicitly declared here.
begin
    for Foo in ... loop
        ...
        <<Foo>> -- Illegal.
        ...
    end loop;
end;
```

12.g/5 {*AI05-0299-1*} {*AI12-0449-1*} The label in this example is hidden from itself by the loop parameter with the same name; the example is illegal. We considered creating a new syntactic category name, separate from direct_name and selector_name, for use in the case of statement labels. However, that would confuse the rules in 8.3, so we didn't do it.

12.1/3 {*AI05-0179-1*} If one or more labels end a sequence_of_statements, an implicit null_statement follows the labels before any following constructs.

12.g.1/3 **Reason:** The semantics of a goto_statement is defined in terms of the statement having (following) that label. Thus we ensure that every label has a following statement, which might be implicit.

Dynamic Semantics

13 The execution of a null_statement has no effect.

14/2 {*AI95-00318-02*} A *transfer of control* is the run-time action of an exit_statement, return statement, goto_statement, or requeue_statement, selection of a terminate_alternative, raising of an exception, or an abort, which causes the next action performed to be one other than what would normally be expected from the other rules of the language. [As explained in 7.6.1, a transfer of control can cause the execution of constructs to be completed and then left, which may trigger finalization.]

The execution of a sequence_of_statements consists of the execution of the individual statements in succession until the sequence_ is completed. 15

Ramification: It could be completed by reaching the end of it, or by a transfer of control. 15.a

{*AI12-0119-1*} Within a parallel construct, if a transfer of control out of the construct is initiated by one of the logical threads of control, an attempt is made to *cancel* all other logical threads of control initiated by the parallel construct. Once all other logical threads of control of the construct either complete or are canceled, the transfer of control occurs. If two or more logical threads of control of the same construct initiate such a transfer of control concurrently, one of them is chosen arbitrarily and the others are canceled. 16/5

{*AI12-0119-1*} When a logical thread of control is canceled, the cancellation causes it to complete as though it had performed a transfer of control to the point where it would have finished its execution. Such a cancellation is deferred while the logical thread of control is executing within an abort-deferred operation (see 9.8), and may be deferred further, but not past a point where the logical thread initiates a new nested parallel construct or reaches an exception handler that is outside such an abort-deferred operation. 17/5

Bounded (Run-Time) Errors

{*AI12-0119-1*} {*AI12-0442-1*} During the execution of a parallel construct, it is a bounded error to invoke an operation that is potentially blocking (see 9.5). Program_Error is raised if the error is detected by the implementation; otherwise, the execution of the potentially blocking operation can either proceed normally, or it can result in the indefinite blocking of some or all of the logical threads of control making up the current task. 18/5

NOTE A statement_identifier that appears immediately within the declarative region of a named loop_statement or an accept_statement is nevertheless implicitly declared immediately within the declarative region of the innermost enclosing body or block_statement; in other words, the expanded name for a named statement is not affected by whether the statement occurs inside or outside a named loop or an accept_statement — only nesting within block_statements is relevant to the form of its expanded name. 19/5

Discussion: Each comment in the following example gives the expanded name associated with an entity declared in the task body: 19.a

```
task body Compute is                                                    19.b
   Sum : Integer := 0;                          -- Compute.Sum
begin
 Outer:                                         -- Compute.Outer
   for I in 1..10 loop      -- Compute.Outer.I
    Blk:                                        -- Compute.Blk
      declare
         Sum : Integer := 0;                    -- Compute.Blk.Sum
      begin
         accept Ent(I : out Integer; J : in Integer) do
                                                -- Compute.Ent.I, Compute.Ent.J
            Compute.Ent.I := Compute.Outer.I;
          Inner:                                -- Compute.Blk.Inner
            for J in 1..10 loop
                                                -- Compute.Blk.Inner.J
               Sum := Sum + Compute.Blk.Inner.J * Compute.Ent.J;
            end loop Inner;
         end Ent;
         Compute.Sum := Compute.Sum + Compute.Blk.Sum;
      end Blk;
   end loop Outer;
   Record_Result(Sum);
end Compute;
```

Examples

Examples of labeled statements: 20

```
<<Here>> <<Ici>> <<Aqui>> <<Hier>> null;                               21
```

22
```
<<After>> X := 1;
```

Extensions to Ada 83

22.a The requeue_statement is new.

Wording Changes from Ada 83

22.b We define the syntactic category statement_identifier to simplify the description. It is used for labels, loop names, and block names. We define the entity associated with the implicit declarations of statement names.

22.c Completion includes completion caused by a transfer of control, although RM83-5.1(6) did not take this view.

Extensions to Ada 95

22.d/2 {*AI95-00318-02*} The extended_return_statement is new (simple_return_statement is merely renamed).

Extensions to Ada 2005

22.e/3 {*AI05-0179-1*} A label can end a sequence_of_statements, eliminating the requirement for having an explicit **null**; statement after an ending label (a common use).

Extensions to Ada 2012

22.f/5 {*AI12-0119-1*} The definition of "parallel constructs" and the parallel_block_statement are new.

5.2 Assignment Statements

1 [An assignment_statement replaces the current value of a variable with the result of evaluating an expression.]

Syntax

2 assignment_statement ::=
*variable*_name := expression;

3 The execution of an assignment_statement includes the evaluation of the expression and the *assignment* of the value of the expression into the *target*. [An assignment operation (as opposed to an assignment_statement) is performed in other contexts as well, including object initialization and by-copy parameter passing.] The *target* of an assignment operation is the view of the object to which a value is being assigned; the target of an assignment_statement is the variable denoted by the *variable*_name.

3.a **Discussion:** Don't confuse this notion of the "target" of an assignment with the notion of the "target object" of an entry call or requeue.

3.b Don't confuse the term "assignment operation" with the assignment_statement. The assignment operation is just one part of the execution of an assignment_statement. The assignment operation is also a part of the execution of various other constructs; see 7.6.1, "Completion and Finalization" for a complete list. Note that when we say, "such-and-such is assigned to so-and-so", we mean that the assignment operation is being applied, and that so-and-so is the target of the assignment operation.

Name Resolution Rules

4/2 {*AI95-00287-01*} The *variable*_name of an assignment_statement is expected to be of any type. The expected type for the expression is the type of the target.

4.a **Implementation Note:** An assignment_statement as a whole is a "complete context", so if the *variable*_name of an assignment_statement is overloaded, the expression can be used to help disambiguate it. For example:

4.b
```
type P1 is access R1;
type P2 is access R2;
```

4.c
```
function F return P1;
function F return P2;
```

4.d
```
X : R1;
begin
  F.all := X;   -- Right hand side helps resolve left hand side
```

Legality Rules

{*AI95-00287-01*} The target [denoted by the *variable*_name] shall be a variable of a nonlimited type. 5/2

If the target is of a tagged class-wide type *T*'Class, then the expression shall either be dynamically tagged, or of type *T* and tag-indeterminate (see 3.9.2). 6

Reason: This is consistent with the general rule that a single dispatching operation shall not have both dynamically tagged and statically tagged operands. Note that for an object initialization (as opposed to the assignment_statement), a statically tagged initialization expression is permitted, since there is no chance for confusion (or Tag_Check failure). Also, in an object initialization, tag-indeterminate expressions of any type covered by *T*'Class would be allowed, but with an assignment_statement, that might not work if the tag of the target was for a type that didn't have one of the dispatching operations in the tag-indeterminate expression. 6.a

Dynamic Semantics

For the execution of an assignment_statement, the *variable*_name and the expression are first evaluated in an arbitrary order. 7

Ramification: Other rules of the language may require that the bounds of the variable be determined prior to evaluating the expression, but that does not necessarily require evaluation of the *variable*_name, as pointed out by the ACID. 7.a

When the type of the target is class-wide: 8

- If the expression is tag-indeterminate (see 3.9.2), then the controlling tag value for the expression is the tag of the target; 9

  Ramification: See 3.9.2, "Dispatching Operations of Tagged Types". 9.a

- Otherwise [(the expression is dynamically tagged)], a check is made that the tag of the value of the expression is the same as that of the target; if this check fails, Constraint_Error is raised. 10

{*AI12-0439-1*} The value of the expression is converted to the subtype of the target. [The conversion can raise an exception (see 4.6).] 11/5

Ramification: 4.6, "Type Conversions" defines what actions and checks are associated with subtype conversion. For non-array subtypes, it is just a constraint check presuming the types match. For array subtypes, it checks the lengths and slides if the target is constrained. "Sliding" means the array doesn't have to have the same bounds, so long as it is the same length. 11.a

{*AI12-0439-1*} In cases involving controlled types, the target is finalized, and an anonymous object can be used as an intermediate in the assignment, as described in 7.6.1, "Completion and Finalization". In any case, the converted value of the expression is then *assigned* to the target, which consists of the following two steps: 12/5

To be honest: 7.6.1 actually says that finalization happens always, but unless controlled types are involved, this finalization during an assignment_statement does nothing. 12.a

- The value of the target becomes the converted value. 13
- {*AI05-0299-1*} {*AI12-0449-1*} If any part of the target is controlled, its value is adjusted as explained in 7.6. 14/5

  Ramification: If any parts of the object are controlled, abort is deferred during the assignment operation itself, but not during the rest of the execution of an assignment_statement. 14.a

NOTE The tag of an object never changes; in particular, an assignment_statement does not change the tag of the target. 15

This paragraph was deleted.{*AI95-00363-01*} 16/2

Ramification: The implicit subtype conversion described above for assignment_statements is performed only for the value of the right-hand side expression as a whole; it is not performed for subcomponents of the value. 16.a

The determination of the type of the variable of an assignment_statement may require consideration of the expression if the variable name can be interpreted as the name of a variable designated by the access value returned by a function call, and similarly, as a component or slice of such a variable (see 8.6, "The Context of Overload Resolution"). 16.b

Examples

17 *Examples of assignment statements:*

18
```
Value := Max_Value - 1;
Shade := Blue;
```

19
```
Next_Frame(F)(M, N) := 2.5;          -- see 4.1.1
U := Dot_Product(V, W);              -- see 6.3
```

20/4
```
{AI12-0056-1} Writer := (Status => Open, Unit => Printer, Line_Count => 60);
                                     -- see 3.8.1
Next.all := (72074, null, Head);     -- see 3.10.1
```

21 *Examples involving scalar subtype conversions:*

22
```
I, J : Integer range 1 .. 10 := 5;
K    : Integer range 1 .. 20 := 15;
 ...
```

23
```
I := J;  -- identical ranges
K := J;  -- compatible ranges
J := K;  -- will raise Constraint_Error if K > 10
```

24 *Examples involving array subtype conversions:*

25
```
A : String(1 .. 31);
B : String(3 .. 33);
 ...
```

26
```
A := B;  -- same number of components
```

27
```
A(1 .. 9)  := "tar sauce";
A(4 .. 12) := A(1 .. 9);   -- A(1 .. 12) = "tartar sauce"
```

28/5 {*AI12-0452-1*} Assignment_statements are well-defined even in the case of overlapping slices of the same array, because the *variable*_name and expression are both evaluated before copying the value into the variable. In the above example, an implementation yielding `A(1 .. 12) = "tartartartar"` would be incorrect.

Extensions to Ada 83

29.a We now allow user-defined finalization and value adjustment actions as part of assignment_statements (see 7.6, “Assignment and Finalization”).

Wording Changes from Ada 83

29.b The special case of array assignment is subsumed by the concept of a subtype conversion, which is applied for all kinds of types, not just arrays. For arrays it provides “sliding”. For numeric types it provides conversion of a value of a universal type to the specific type of the target. For other types, it generally has no run-time effect, other than a constraint check.

29.c We now cover in a general way in 3.7.2 the erroneous execution possible due to changing the value of a discriminant when the variable in an assignment_statement is a subcomponent that depends on discriminants.

Incompatibilities With Ada 95

29.d/2 {*AI95-00287-01*} The change of the limited check from a resolution rule to a legality rule is not quite upward compatible. For example

29.e/2
```
type AccNonLim is access NonLim;
function Foo (Arg : in Integer) return AccNonLim;
type AccLim is access Lim;
function Foo (Arg : in Integer) return AccLim;
Foo(2).all := Foo(1).all;
```

29.f/2 where NonLim is a nonlimited type and Lim is a limited type. The assignment is legal in Ada 95 (only the first Foo would be considered), and is ambiguous in Ada 2005. We made the change because we want limited types to be as similar to nonlimited types as possible. Limited expressions are now allowed in all other contexts (with a similar incompatibility), and it would be odd if assignments had different resolution rules (which would eliminate ambiguities in some cases). Moreover, examples like this one are rare, as they depend on assigning into overloaded function calls.

5.2.1 Target Name Symbols

{*AI12-0125-3*} @, known as the *target name* of an assignment statement, provides an abbreviation to avoid repetition of potentially long names in assignment statements. 1/5

Syntax

{*AI12-0125-3*} target_name ::= @ 2/5

Name Resolution Rules

{*AI12-0125-3*} [If a target_name occurs in an assignment_statement *A*, the *variable*_name *V* of *A* is a complete context. The target name is a constant view of *V*, having the nominal subtype of *V*.] 3/5

Proof: {*AI12-0125-3*} {*AI12-0322-1*} The complete context rule is formally given in 8.6. The constant view rule is formally given in 3.3; the nominal subtype is a property taken from the target object as described below in Dynamic Semantics. 3.a/5

Legality Rules

{*AI12-0125-3*} {*AI12-0322-1*} A target_name shall appear only in the expression of an assignment_statement. 4/5

Dynamic Semantics

{*AI12-0125-3*} {*AI12-0322-1*} {*AI12-0449-1*} For the execution of an assignment_statement with one or more target_names appearing in its expression, the *variable*_name *V* of the assignment_statement is evaluated first to determine the object denoted by *V*, and then the expression of the assignment_statement is evaluated with the evaluation of each target_name yielding a constant view of the target whose properties are otherwise identical to those of the view provided by *V*. The remainder of the execution of the assignment_statement is as given in 5.2. 5/5

To be honest: The properties here include static properties like whether the target_name is aliased and the nominal subtype of the target_name. It was too weird to give separate rules for static and dynamic properties that said almost the same thing. 5.a/5

Ramification: {*AI12-0322-1*} Use of a target_name can be erroneous if the *variable*_name *V* is a discriminant-dependent component, and some other constituent of the expression modifies the discriminant governing the component *V*. The assignment probably would be erroneous anyway, but the use of a target_name eliminates the possibility that a later evaluation of *V* raises an exception before any erroneous execution occurs. See 3.7.2. 5.b/5

Examples

{*AI12-0429-1*} *Examples of the use of target name symbols:* 6/5

```
{AI12-0125-3} {AI12-0379-1} {AI12-0442-1} Board(1, 1) := @ + 1.0;                7/5
              -- An abbreviation for Board(1, 1) := Board(1, 1) + 1.0;
              -- (Board is declared in 3.6.1).

{AI12-0125-3} My_Complex_Array : array (1 .. Max) of Complex; -- See 3.3.2, 3.8.  8/5
...
-- Square the element in the Count (see 3.3.1) position:
My_Complex_Array (Count) := (Re => @.Re**2 - @.Im**2,
                             Im => 2.0 * @.Re * @.Im);
   -- A target_name can be used multiple times and
   -- as a prefix if desired.
```

Extensions to Ada 2012

{*AI12-0125-3*} {*AI12-0322-1*} The target name symbol @ is new. 8.a/5

5.3 If Statements

1 [An if_statement selects for execution at most one of the enclosed sequences_of_statements, depending on the (truth) value of one or more corresponding conditions.]

Syntax

2
```
if_statement ::=
   if condition then
     sequence_of_statements
  {elsif condition then
     sequence_of_statements}
  [else
     sequence_of_statements]
   end if;
```

Paragraphs 3 and 4 were deleted.

Dynamic Semantics

5/3 {*AI05-0264-1*} For the execution of an if_statement, the condition specified after **if**, and any conditions specified after **elsif**, are evaluated in succession (treating a final **else** as **elsif** True **then**), until one evaluates to True or all conditions are evaluated and yield False. If a condition evaluates to True, then the corresponding sequence_of_statements is executed; otherwise, none of them is executed.

5.a **Ramification:** The part about all evaluating to False can't happen if there is an **else**, since that is herein considered equivalent to **elsif** True **then**.

Examples

6 *Examples of if statements:*

7
```
if Month = December and Day = 31 then
   Month := January;
   Day   := 1;
   Year  := Year + 1;
end if;
```

8
```
if Line_Too_Short then
   raise Layout_Error;
elsif Line_Full then
   New_Line;
   Put(Item);
else
   Put(Item);
end if;
```

9
```
if My_Car.Owner.Vehicle /= My_Car then          -- see 3.10.1
   Report ("Incorrect data");
end if;
```

Wording Changes from Ada 2005

9.a/3 {*AI05-0147-1*} Moved definition of condition to 4.5.7 in order to eliminate a forward reference.

5.4 Case Statements

1 [A case_statement selects for execution one of a number of alternative sequences_of_statements; the chosen alternative is defined by the value of an expression.]

Syntax

{*AI05-0188-1*} case_statement ::= 2/3
  case *selecting*_expression **is**
    case_statement_alternative
    {case_statement_alternative}
  end case;

case_statement_alternative ::= 3
  when discrete_choice_list =>
    sequence_of_statements

Name Resolution Rules

{*AI05-0188-1*} The *selecting*_expression is expected to be of any discrete type. The expected type for each discrete_choice is the type of the *selecting*_expression. 4/3

Legality Rules

{*AI05-0153-3*} The choice_expressions, subtype_indications, and ranges given as discrete_choices of a case_statement shall be static. [A discrete_choice **others**, if present, shall appear alone and in the last discrete_choice_list.] 5/3

{*AI05-0188-1*} {*AI05-0240-1*} The possible values of the *selecting*_expression shall be covered (see 3.8.1) as follows: 6/3

Discussion: {*AI05-0240-1*} The meaning of "covered" here and in the following rules is that of the term "cover a value" that is defined in 3.8.1. 6.a/3

- {*AI05-0003-1*} {*AI05-0153-3*} {*AI05-0188-1*} {*AI05-0262-1*} {*AI12-0071-1*} If the *selecting*_expression is a name [(including a type_conversion, qualified_expression, or function_call)] having a static and constrained nominal subtype, then each non-**others** discrete_choice shall cover only values in that subtype that satisfy its predicates (see 3.2.4), and each value of that subtype that satisfies its predicates shall be covered by some discrete_choice [(either explicitly or by **others**)]. 7/4

  Ramification: Although not official names of objects, a value conversion still has a defined nominal subtype, namely its target subtype. See 4.6. 7.a

- {*AI05-0188-1*} If the type of the *selecting*_expression is *root_integer*, *universal_integer*, or a descendant of a formal scalar type, then the case_statement shall have an **others** discrete_choice. 8/3

  Reason: This is because the base range is implementation defined for *root_integer* and *universal_integer*, and not known statically in the case of a formal scalar type. 8.a

- {*AI05-0188-1*} Otherwise, each value of the base range of the type of the *selecting*_expression shall be covered [(either explicitly or by **others**)]. 9/3

Two distinct discrete_choices of a case_statement shall not cover the same value. 10

Ramification: {*AI05-0188-1*} The goal of these coverage rules is that any possible value of the *selecting*_expression of a case_statement should be covered by exactly one discrete_choice of the case_statement, and that this should be checked at compile time. The goal is achieved in most cases, but there are two minor loopholes: 10.a/3

- If the expression reads an object with an invalid representation (e.g. an uninitialized object), then the value can be outside the covered range. This can happen for static constrained subtypes, as well as nonstatic or unconstrained subtypes. It cannot, however, happen if the case_statement has the discrete_choice **others**, because **others** covers all values, even those outside the subtype. 10.b

- {*AI95-00114-01*} {*AI05-0188-1*} If the compiler chooses to represent the value of an expression of an unconstrained subtype in a way that includes values outside the bounds of the subtype, then those values can be outside the covered range. For example, if X: Integer := Integer'Last;, and the case *selecting*_expression is X+1, then the implementation might choose to produce the correct value, which is outside the bounds of 10.c/3

Integer. (It might raise Constraint_Error instead.) This case can only happen for nongeneric subtypes that are either unconstrained or nonstatic (or both). It can only happen if there is no **others** discrete_choice.

10.d In the uninitialized variable case, the value might be anything; hence, any alternative can be chosen, or Constraint_Error can be raised. (We intend to prevent, however, jumping to random memory locations and the like.) In the out-of-range case, the behavior is more sensible: if there is an **others**, then the implementation may choose to raise Constraint_Error on the evaluation of the expression (as usual), or it may choose to correctly evaluate the expression and therefore choose the **others** alternative. Otherwise (no **others**), Constraint_Error is raised either way — on the expression evaluation, or for the case_statement itself.

10.e For an enumeration type with a discontiguous set of internal codes (see 13.4), the only way to get values in between the proper values is via an object with an invalid representation; there is no "out-of-range" situation that can produce them.

Dynamic Semantics

11/3 {*AI05-0188-1*} For the execution of a case_statement, the *selecting*_expression is first evaluated.

12/3 {*AI05-0188-1*} If the value of the *selecting*_expression is covered by the discrete_choice_list of some case_statement_alternative, then the sequence_of_statements of the _alternative is executed.

13 Otherwise (the value is not covered by any discrete_choice_list, perhaps due to being outside the base range), Constraint_Error is raised.

13.a/5 **Ramification:** {*AI12-0005-1*} In this case, the value fails to satisfy its (static) predicate (possible when the predicate is disabled), is outside the base range of its type, or is an invalid representation.

14/5 NOTE {*AI12-0440-1*} The execution of a case_statement chooses one and only one alternative. Qualification of the expression of a case_statement by a static subtype can often be used to limit the number of choices that can be given explicitly.

Examples

15 *Examples of case statements:*

16
```
case Sensor is
   when Elevation => Record_Elevation(Sensor_Value);
   when Azimuth   => Record_Azimuth  (Sensor_Value);
   when Distance  => Record_Distance (Sensor_Value);
   when others    => null;
end case;
```

17
```
case Today is
   when Mon       => Compute_Initial_Balance;
   when Fri       => Compute_Closing_Balance;
   when Tue .. Thu=> Generate_Report(Today);
   when Sat .. Sun=> null;
end case;
```

18
```
case Bin_Number(Count) is
   when 1    => Update_Bin(1);
   when 2    => Update_Bin(2);
   when 3 | 4 =>
      Empty_Bin(1);
      Empty_Bin(2);
   when others  => raise Error;
end case;
```

Incompatibilities With Ada 83

18.a.1/1 In Ada 95, function_calls and type_conversions are names, whereas in Ada 83, they were expressions. Therefore, if the expression of a case_statement is a function_call or type_conversion, and the result subtype is static, it is illegal to specify a choice outside the bounds of the subtype. For this case in Ada 83 choices only are required to be in the base range of the type.

18.a.2/1 In addition, the rule about which choices must be covered is unchanged in Ada 95. Therefore, for a case_statement whose expression is a function_call or type_conversion, Ada 83 required covering all choices in the base range, while Ada 95 only requires covering choices in the bounds of the subtype. If the case_statement does not include an **others** discrete_choice, then a legal Ada 83 case_statement will be illegal in Ada 95 if the bounds of the subtype are different than the bounds of the base type.

Extensions to Ada 83

In Ada 83, the expression in a case_statement is not allowed to be of a generic formal type. This restriction is removed in Ada 95; an **others** discrete_choice is required instead. 18.a

In Ada 95, a function call is the name of an object; this was not true in Ada 83 (see 4.1, “Names”). This change makes the following case_statement legal: 18.b

```
subtype S is Integer range 1..2;
function F return S;
case F is
   when 1 => ...;
   when 2 => ...;
   -- No others needed.
end case;
```
18.c

{*AI05-0005-1*} Note that the result subtype given in a function renaming_declaration is ignored; for a case_statement whose expression calls a such a function, the full coverage rules are checked using the result subtype of the original function. Note that predefined operators such as "+" have an unconstrained result subtype (see 4.5.1). Note that generic formal functions do not have static result subtypes. Note that the result subtype of an inherited subprogram need not correspond to any nameable subtype; there is still a perfectly good result subtype, though. 18.d/3

Wording Changes from Ada 83

Ada 83 forgot to say what happens for “legally” out-of-bounds values. 18.e

We take advantage of rules and terms (e.g. *cover a value*) defined for discrete_choices and discrete_choice_lists in 3.8.1, “Variant Parts and Discrete Choices”. 18.f

In the Name Resolution Rule for the case expression, we no longer need RM83-5.4(3)'s “which must be determinable independently of the context in which the expression occurs, but using the fact that the expression must be of a discrete type”, because the expression is now a complete context. See 8.6, “The Context of Overload Resolution”. 18.g

Since type_conversions are now defined as names, their coverage rule is now covered under the general rule for names, rather than being separated out along with qualified_expressions. 18.h

Wording Changes from Ada 2005

{*AI05-0003-1*} Rewording to reflect that a qualified_expression is now a name. 18.i/3

{*AI05-0153-3*} Revised for changes to discrete_choices made to allow static predicates (see 3.2.4) as case choices (see 3.8.1). 18.j/3

{*AI05-0188-1*} Added the *selecting_* prefix to make this wording consistent with case_expression, and to clarify which expression is being talked about in the wording. 18.k/3

Wording Changes from Ada 2012

{*AI12-0071-1*} **Corrigendum:** Updated wording of case coverage to use the new term "satisfies the predicates" (see 3.2.4). 18.l/4

5.5 Loop Statements

{*AI12-0119-1*} [A loop_statement includes a sequence_of_statements that is to be executed repeatedly, zero or more times with the iterations running sequentially or concurrently with one another.] 1/5

Syntax

loop_statement ::= 2
  [*loop_*statement_identifier:]
    [iteration_scheme] **loop**
      sequence_of_statements
    end loop [*loop_*identifier];

{*AI05-0139-2*} {*AI12-0119-1*} {*AI12-0189-1*} {*AI12-0251-1*} {*AI12-0266-1*} {*AI12-0326-2*} {*AI12-0355-2*} iteration_scheme ::= **while** condition 3/5
  | **for** loop_parameter_specification
  | **for** iterator_specification

| [**parallel** [aspect_specification]]
for procedural_iterator
| **parallel** [(chunk_specification)] [aspect_specification]
for loop_parameter_specification
| **parallel** [(chunk_specification)] [aspect_specification]
for iterator_specification

3.1/5 {*AI12-0251-1*} chunk_specification ::=
*integer*_simple_expression
| defining_identifier **in** discrete_subtype_definition

4/5 {*AI12-0250-1*} loop_parameter_specification ::=
defining_identifier **in** [**reverse**] discrete_subtype_definition
[iterator_filter]

4.1/5 {*AI12-0250-1*} iterator_filter ::= **when** condition

5 If a loop_statement has a *loop*_statement_identifier, then the identifier shall be repeated after the **end loop**; otherwise, there shall not be an identifier after the **end loop**.

5.1/5 {*AI12-0119-1*} An iteration_scheme that begins with the reserved word **parallel** shall not have the reserved word **reverse** in its loop_parameter_specification.

Name Resolution Rules

5.2/5 {*AI12-0251-1*} In a chunk_specification that is an *integer*_simple_expression, the *integer*_simple_expression is expected to be of any integer type.

Static Semantics

6/5 {*AI12-0061-1*} A loop_parameter_specification declares a *loop parameter*, which is an object whose subtype (and nominal subtype) is that defined by the discrete_subtype_definition.

6.1/5 {*AI12-0251-1*} In a chunk_specification that has a discrete_subtype_definition, the chunk_specification declares a *chunk parameter* object whose subtype (and nominal subtype) is that defined by the discrete_subtype_definition.

Dynamic Semantics

6.2/5 {*AI12-0250-1*} {*AI12-0327-1*} The *filter* of an *iterator construct* (a loop_parameter_specification, iterator_specification, or procedural_iterator) is defined to be *satisfied* when there is no iterator_filter for the iterator construct, or when the condition of the iterator_filter evaluates to True for a given iteration of the iterator construct.

6.a/5 **Term entry: iterator filter** — construct that is used to restrict the elements produced by an iteration to those for which a boolean condition evaluates to True

6.3/5 {*AI12-0250-1*} {*AI12-0327-1*} If a sequence_of_statements of a loop_statement with an iterator construct is said to be *conditionally executed*, then the statements are executed only when the filter of the iterator construct is satisfied.

6.4/5 {*AI12-0250-1*} {*AI12-0327-1*} The loop iterators loop_parameter_specification and iterator_specification can also be used in contexts other than loop_statements (for example, see 4.3.5 and 4.5.8). In such a context, the iterator *conditionally produces* values in the order specified for the associated construct below or in 5.5.2. The values produced are the values given to the loop parameter when the filter of the iterator construct is satisfied for that value. [No value is produced when the condition of an iterator_filter evaluates to False.]

{*AI12-0119-1*} For the execution of a loop_statement, the sequence_of_statements is executed zero or more times, until the loop_statement is complete. The loop_statement is complete when a transfer of control occurs that transfers control out of the loop, or, in the case of an iteration_scheme, as specified below. 7/5

For the execution of a loop_statement with a **while** iteration_scheme, the condition is evaluated before each execution of the sequence_of_statements; if the value of the condition is True, the sequence_of_statements is executed; if False, the execution of the loop_statement is complete. 8

{*AI12-0251-1*} {*AI12-0294-1*} If the reserved word **parallel** is present in the iteration_scheme of a loop_statement (a *parallel loop*), the iterations are partitioned into one or more *chunks*, each with its own separate logical thread of control (see Clause 9). If a chunk_specification is present in a parallel loop, it is elaborated first, and the result of the elaboration determines the maximum number of chunks used for the parallel loop. If the chunk_specification is an *integer*_simple_expression, the elaboration evaluates the expression, and the value of the expression determines the maximum number of chunks. If a discrete_subtype_definition is present, the elaboration elaborates the discrete_subtype_definition, which defines the subtype of the chunk parameter, and the number of values in this subtype determines the maximum number of chunks. After elaborating the chunk_specification, a check is made that the determined maximum number of chunks is greater than zero. If this check fails, Program_Error is raised. 8.1/5

{*AI05-0139-2*} {*AI05-0262-1*} {*AI12-0071-1*} {*AI12-0119-1*} {*AI12-0250-1*} {*AI12-0251-1*} {*AI12-0294-1*} {*AI12-0355-2*} {*AI12-0416-1*} For the execution of a loop_statement that has an iteration_scheme including a loop_parameter_specification, after elaborating the chunk_specification and aspect_specification, if any, the loop_parameter_specification is elaborated. This elaborates the discrete_subtype_definition, which defines the subtype of the loop parameter. If the discrete_subtype_definition defines a subtype with a null range, the execution of the loop_statement is complete. Otherwise, the sequence_of_statements is conditionally executed once for each value of the discrete subtype defined by the discrete_subtype_definition that satisfies the predicates of the subtype (or until the loop is left as a consequence of a transfer of control). Prior to each such iteration, the corresponding value of the discrete subtype is assigned to the loop parameter associated with the given iteration. If the loop is a parallel loop, each chunk has its own logical thread of control with its own copy of the loop parameter; otherwise (a *sequential loop*), a single logical thread of control performs the loop, and there is a single copy of the loop parameter. Each logical thread of control handles a distinct subrange of the values of the subtype of the loop parameter such that all values are covered with no overlaps. Within each logical thread of control, the values are assigned to the loop parameter in increasing order unless the reserved word **reverse** is present, in which case the values are assigned in decreasing order. In the absence of a transfer of control, the associated parallel construct of a loop_parameter_specification is complete when all of its logical threads of control are complete. 9/5

To be honest: {*AI12-0294-1*} This wording does not describe when the loop parameter object(s) are created. That creation has no side-effects (other than possibly raising Storage_Error, but anything can do that), so we simplified the wording by leaving it out. Each object has to be created before any iteration that depends on it starts, but we do not (for instance) require that the objects are all created at once at the start of the loop, nor that the objects are created after the elaboration of the discrete_subtype_definition. 9.a.1/5

This paragraph was deleted.{*AI12-0294-1*} 9.a/5

Ramification: {*AI05-0262-1*} The predicate (if any) necessarily has to be a static predicate as a dynamic predicate is explicitly disallowed — see 3.2.4. 9.b/3

Reason: {*AI05-0262-1*} If there is a predicate, the loop still visits the values in the order of the underlying base type; the order of the values in the predicate is irrelevant. This is the case so that the following loops have the same sequence of calls and parameters on procedure Call for any subtype S: 9.c/3

```
for I in S loop                                                    9.d
   Call (I);
end loop;
```

9.e
```
for I in S'Base loop
   if I in S then
      Call (I);
   end if;
end loop;
```

9.f/5 **Discussion:** {*AI12-0416-1*} The rules for completing a parallel construct when there is a transfer of control are given in 5.1.

10/5 {*AI12-0251-1*} {*AI12-0294-1*} If a chunk_specification with a discrete_subtype_definition is present, then the logical thread of control associated with a given chunk has its own copy of the chunk parameter initialized with a distinct value from the discrete subtype defined by the discrete_subtype_definition. The values of the chunk parameters are assigned such that they increase with increasing values of the ranges covered by the corresponding loop parameters.

11/5 {*AI12-0251-1*} Whether or not a chunk_specification is present in a parallel loop, the total number of iterations of the loop represents an upper bound on the number of logical threads of control devoted to the loop.

12/5 {*AI05-0262-1*} {*AI12-0250-1*} {*AI12-0266-1*} [For details about the execution of a loop_statement with the iteration_scheme including an iterator_specification, see 5.5.2. For details relating to a procedural_iterator, see 5.5.3.]

13/5 NOTE 1 {*AI12-0250-1*} A loop parameter declared by a loop_parameter_specification is a constant; it cannot be updated within the sequence_of_statements of the loop (see 3.3).

14/5 NOTE 2 {*AI12-0442-1*} No separate object_declaration is expected for a loop parameter, since the loop parameter is automatically declared by the loop_parameter_specification. The scope of a loop parameter extends from the loop_parameter_specification to the end of the loop_statement, and the visibility rules are such that a loop parameter is only visible within the sequence_of_statements of the loop.

14.a **Implementation Note:** An implementation could give a warning if a variable is hidden by a loop_parameter_specification.

15/5 NOTE 3 The discrete_subtype_definition of a for loop is elaborated just once. Use of the reserved word **reverse** does not alter the discrete subtype defined, so that the following iteration_schemes are not equivalent; the first has a null range.

16
```
for J in reverse 1 .. 0
for J in 0 .. 1
```

16.a **Ramification:** If a loop_parameter_specification has a static discrete range, the subtype of the loop parameter is static.

Examples

17 *Example of a loop statement without an iteration scheme:*

18
```
loop
   Get(Current_Character);
   exit when Current_Character = '*';
end loop;
```

19 *Example of a loop statement with a **while** iteration scheme:*

20
```
while Bid(N).Price < Cut_Off.Price loop
   Record_Bid(Bid(N).Price);
   N := N + 1;
end loop;
```

21 *Example of a loop statement with a **for** iteration scheme:*

22
```
for J in Buffer'Range loop      -- works even with a null range
   if Buffer(J) /= Space then
      Put(Buffer(J));
   end if;
end loop;
```

Example of a loop statement with a name: 23

```
Summation:                                                              24
   while Next /= Head loop        -- see 3.10.1
      Sum  := Sum + Next.Value;
      Next := Next.Succ;
   end loop Summation;
```

{*AI12-0119-1*} {*AI12-0312-1*} *Example of a simple parallel loop:* 25/5

```
-- see 3.6                                                              26/5
parallel
for I in Grid'Range(1) loop
   Grid(I, 1) := (for all J in Grid'Range(2) => Grid(I,J) = True);
end loop;
```

{*AI12-0312-1*} *Example of a parallel loop with a chunk specification:* 27/5

```
declare                                                                 28/5
   subtype Chunk_Number is Natural range 1 .. 8;

   Partial_Sum,                                                         29/5
   Partial_Max : array (Chunk_Number) of Natural := (others => 0);
   Partial_Min : array (Chunk_Number) of Natural :=
                       (others => Natural'Last);

begin                                                                   30/5
   parallel (Chunk in Chunk_Number)
   for I in Grid'Range(1) loop
      declare
         True_Count : constant Natural :=
           [for J in Grid'Range(2) =>
              (if Grid (I, J) then 1 else 0)]'Reduce("+",0);
      begin
         Partial_Sum (Chunk) := @ + True_Count;
         Partial_Min (Chunk) := Natural'Min(@, True_Count);
         Partial_Max (Chunk) := Natural'Max(@, True_Count);
      end;
   end loop;

{AI12-0386-1}    Put_Line                                              31/5
     ("Total=" & Partial_Sum'Reduce("+", 0)'Image &
      ", Min=" & Partial_Min'Reduce(Natural'Min, Natural'Last)'Image &
      ", Max=" & Partial_Max'Reduce(Natural'Max, 0)'Image);
end;
```

{*AI12-0312-1*} *For an example of an* iterator_filter*, see 4.5.8.* 32/5

Wording Changes from Ada 83

The constant-ness of loop parameters is specified in 3.3, “Objects and Named Numbers”. 32.a

Wording Changes from Ada 2005

{*AI05-0139-2*} {*AI05-0262-1*} {*AI05-0299-1*} Generalized iterator_specifications are allowed in **for** loops; these are documented as an extension in the appropriate subclause. 32.b/3

Extensions to Ada 2012

{*AI12-0119-1*} {*AI12-0251-1*} {*AI12-0266-1*} {*AI12-0294-1*} Parallel loops are new. 32.c/5

{*AI12-0250-1*} An iterator_filter is now allowed on loop_parameter_specifications. This is mainly for consistency with aggregate and reduction iterators, where it eliminates the need for temporary objects. 32.d/5

Wording Changes from Ada 2012

{*AI12-0071-1*} **Corrigendum:** Updated wording of loop execution to use the new term "satisfies the predicates" (see 3.2.4). 32.e/4

{*AI12-0061-1*} Added text so that the nominal subtype of a loop parameter is clearly defined. 32.f/5

5.5.1 User-Defined Iterator Types

Static Semantics

1/3 {*AI05-0139-2*} The following language-defined generic library package exists:

2/5
```
{AI12-0241-1} generic
   type Cursor;
   with function Has_Element (Position : Cursor) return Boolean;
package Ada.Iterator_Interfaces
   with Pure, Nonblocking => False is

3/3    type Forward_Iterator is limited interface;
   function First (Object : Forward_Iterator) return Cursor is abstract;
   function Next (Object : Forward_Iterator; Position : Cursor)
      return Cursor is abstract;

4/3    type Reversible_Iterator is limited interface and Forward_Iterator;
   function Last (Object : Reversible_Iterator) return Cursor is abstract;
   function Previous (Object : Reversible_Iterator; Position : Cursor)
      return Cursor is abstract;

4.1/5  {AI12-0266-1}    type Parallel_Iterator is limited interface and Forward_Iterator;

4.2/5  {AI12-0266-1}    subtype Chunk_Index is Positive;

4.3/5  {AI12-0266-1}    function Is_Split (Object : Parallel_Iterator)
         return Boolean is abstract;

4.4/5  {AI12-0266-1}    procedure Split_Into_Chunks (Object     : in out
       Parallel_Iterator;
                                    Max_Chunks : in     Chunk_Index) is abstract
         with Pre'Class   => not Object.Is_Split or else raise Program_Error,
              Post'Class  => Object.Is_Split and then
                             Object.Chunk_Count <= Max_Chunks;

4.5/5  {AI12-0266-1}    function Chunk_Count (Object : Parallel_Iterator)
         return Chunk_Index is abstract
         with Pre'Class   => Object.Is_Split or else raise Program_Error;

4.6/5  {AI12-0266-1}    function First (Object : Parallel_Iterator;
                          Chunk  : Chunk_Index) return Cursor is abstract
         with Pre'Class   => (Object.Is_Split and then
                                 Chunk <= Object.Chunk_Count)
                             or else raise Program_Error;

4.7/5  {AI12-0266-1}    function Next (Object   : Parallel_Iterator;
                         Position : Cursor;
                         Chunk    : Chunk_Index) return Cursor is abstract
         with Pre'Class   => (Object.Is_Split and then
                                 Chunk <= Object.Chunk_Count)
                             or else raise Program_Error;

4.8/5  {AI12-0266-1}    type Parallel_Reversible_Iterator is limited interface
         and Parallel_Iterator and Reversible_Iterator;

5/3    end Ada.Iterator_Interfaces;
```

5.a/5 **Reason:** {*AI12-0241-1*} This package must allow blocking (Nonblocking => False) for compatibility. The purpose of this package is to provide a template for overriding user-defined routines; and such routines can only allow blocking if the root type does so. Users can still declare their overridding routines nonblocking if they wish.

6/5 {*AI05-0139-2*} {*AI12-0266-1*} An *iterator type* is a type descended from the Forward_Iterator interface from some instance of Ada.Iterator_Interfaces. A *reversible iterator type* is a type descended from the Reversible_Iterator interface from some instance of Ada.Iterator_Interfaces. A *parallel iterator type* is a type descended from the Parallel_Iterator interface from some instance of Ada.Iterator_Interfaces. A type descended from the Parallel_Reversible_Iterator interface from some instance of Ada.Iterator_Interfaces is both a parallel iterator type and a reversible iterator type. An *iterator object* is an object of an iterator type. A *reversible iterator object* is an object of a reversible iterator type. A *parallel iterator object* is an

object of a parallel iterator type. The formal subtype Cursor from the associated instance of Ada.Iterator_Interfaces is the *iteration cursor subtype* for the iterator type.

{*AI05-0139-2*} {*AI05-0292-1*} The following type-related operational aspects may be specified for an indexable container type *T* (see 4.1.6): 7/3

{*AI12-0111-1*} {*AI12-0428-1*} Default_Iterator 8/5

This aspect is specified by a name that denotes exactly one function declared immediately within the same declaration list in which *T*, or the declaration completed by *T*, is declared, whose first parameter is of type *T* or *T*'Class or an access parameter whose designated type is type *T* or *T*'Class, whose other parameters, if any, have default expressions, and whose result type is an iterator type. This function is the *default iterator function* for *T*. Its result subtype is the *default iterator subtype* for *T*. The iteration cursor subtype for the default iterator subtype is the *default cursor subtype* for *T*. This aspect is inherited by descendants of type *T* (including *T*'Class).

Aspect Description for Default_Iterator: Default iterator to be used in **for** loops. 8.a/3

{*AI12-0111-1*} Iterator_Element 9/5

This aspect is specified by a name that denotes a subtype. This is the *default element subtype* for *T*. This aspect is inherited by descendants of type *T* (including *T*'Class).

Aspect Description for Iterator_Element: Element type to be used for user-defined iterators. 9.a/3

{*AI12-0111-1*} Iterator_View 9.1/5

This aspect is specified by a name that denotes a type *T2* with the following properties:

- *T2* is declared in the same compilation unit as *T*; 9.2/5
- *T2* is an iterable container type; 9.3/5
- *T2* has a single discriminant which is an access discriminant designating *T*; and 9.4/5
- The default iterator subtypes for *T* and *T2* statically match. 9.5/5

This aspect is never inherited[, even by *T*'Class]. 9.6/5

Reason: Iterator_View allows specifying an alternative type to be automatically used by container element iterators; see 5.5.2. This allows setting state for an iteration only once rather than for each individual reference. 9.b/5

Ramification: Since Iterator_View is not inherited, it does not apply to *T*'Class. Otherwise, the type of the iterator object would not be known at compile-time (since it necessarily has to be different for each descendant). 9.c/5

Aspect Description for Iterator_View: An alternative type to used for container element iterators. 9.d/5

This paragraph was deleted.{*AI12-0111-1*} 10/5

{*AI05-0139-2*} {*AI05-0292-1*} {*AI12-0266-1*} An *iterable container type* is an indexable container type with specified Default_Iterator and Iterator_Element aspects. A *reversible iterable container type* is an iterable container type with the default iterator type being a reversible iterator type. A *parallel iterable container type* is an iterable container type with the default iterator type being a parallel iterator type. An *iterable container object* is an object of an iterable container type. A *reversible iterable container object* is an object of a reversible iterable container type. A *parallel iterable container object* is an object of a parallel iterable container type. 11/5

Term entry: iterable container type — type that has user-defined behavior for iteration, via the Default_Iterator and Iterator_Element aspects 11.a/5

{*AI12-0138-1*} The Default_Iterator and Iterator_Element aspects are nonoverridable (see 13.1.1). 11.1/4

Reason: This ensures that all descendants of an iterable container type have aspects with the same properties. This prevents generic contract problems with formal derived types. 11.b/4

Legality Rules

12/3 {*AI05-0139-2*} {*AI05-0292-1*} The Constant_Indexing aspect (if any) of an iterable container type *T* shall denote exactly one function with the following properties:

13/3 • the result type of the function is covered by the default element type of *T* or is a reference type (see 4.1.5) with an access discriminant designating a type covered by the default element type of *T*;

14/3 • the type of the second parameter of the function covers the default cursor type for *T*;

15/3 • if there are more than two parameters, the additional parameters all have default expressions.

16/3 This function (if any) is the *default constant indexing function* for *T*.

16.a/3 **Ramification:** This does not mean that Constant_Indexing has to designate only one subprogram, only that there is only one routine that meets all of these properties. There can be other routines designated by Constant_Indexing, but they cannot have the profile described above. For instance, map containers have a version of Constant_Indexing that takes a key instead of a cursor; this is allowed.

17/3 {*AI05-0139-2*} {*AI05-0292-1*} The Variable_Indexing aspect (if any) of an iterable container type *T* shall denote exactly one function with the following properties:

18/3 • the result type of the function is a reference type (see 4.1.5) with an access discriminant designating a type covered by the default element type of *T*;

19/3 • the type of the second parameter of the function covers the default cursor type for *T*;

20/3 • if there are more than two parameters, the additional parameters all have default expressions.

21/3 This function (if any) is the *default variable indexing function* for *T*.

Erroneous Execution

22/5 {*AI12-0354-1*} A call on the First or Next operation on a given Parallel_Iterator object with a given Chunk value, which does not propagate an exception, should return a Cursor value that either yields False when passed to Has_Element, or that identifies an element distinct from any Cursor value returned by a call on a First or Next operation on the same Parallel_Iterator object with a different Chunk value. If the First or Next operations with a Chunk parameter behave in any other manner, execution is erroneous.

22.a/5 **Reason:** This describes the expectations from a user-written parallel iterator. If the expectations are not met, execution is erroneous so that implementations do not need to go to heroic efforts to avoid problems caused by bad iterators. This is similar to the handling of storage pools, see 13.11.

Extensions to Ada 2005

22.b/3 {*AI05-0139-2*} User-defined iterator types are new in Ada 2012.

Incompatibilities With Ada 2012

22.c/4 {*AI12-0138-1*} **Corrigendum:** Defined Default_Iterator and Iterator_Element to be nonoveridable, which makes redefinitions and hiding of these aspects illegal. It's possible that some program could violate one of these new restrictions, but in most cases this can easily be worked around by using overriding rather than redefinition.

22.d/5 {*AI12-0266-1*} Various new types and subprograms are newly added to Ada.Iterator_Interfaces. Therefore, a use clause conflict is possible; see the introduction of Annex A for more on this topic.

Extensions to Ada 2012

22.e/5 {*AI12-0111-1*} Aspect Iterator_View is new; it allows container element iterators to set the tampering state once rather than for each use of the element.

22.f/5 {*AI12-0266-1*} {*AI12-0354-1*} Parallel iterator interfaces are new; they allow user-defined parallel loops to be defined.

5.5.2 Generalized Loop Iteration

{*AI05-0139-2*} Generalized forms of loop iteration are provided by an iterator_specification. 1/3

Syntax

{*AI05-0139-2*} {*AI05-0292-1*} {*AI12-0156-1*} {*AI12-0250-1*} iterator_specification ::= 2/5
defining_identifier [: loop_parameter_subtype_indication] **in** [**reverse**] *iterator*_name [iterator_filter]
| defining_identifier [: loop_parameter_subtype_indication] **of** [**reverse**] *iterable*_name [iterator_filter]

{*AI12-0156-1*} loop_parameter_subtype_indication ::= subtype_indication | access_definition 2.1/5

{*AI12-0266-1*} If an iterator_specification is for a parallel construct, the reserved word **reverse** shall not appear in the iterator_specification. 2.2/5

Name Resolution Rules

{*AI05-0139-2*} {*AI05-0292-1*} For the first form of iterator_specification, called a *generalized iterator*, the expected type for the *iterator*_name is any iterator type. For the second form of iterator_specification, the expected type for the *iterable*_name is any array or iterable container type. If the *iterable*_name denotes an array object, the iterator_specification is called an *array component iterator*; otherwise it is called a *container element iterator*. 3/3

Term entry: iterator — construct that is used to loop over the elements of an array or container 3.a/5
Note: Iterators can be user defined, and can perform arbitrary computations to access elements from a container.

Legality Rules

{*AI05-0139-2*} {*AI12-0266-1*} If the reserved word **reverse** appears, the iterator_specification is a *reverse iterator*. If the iterator_specification is for a parallel construct, the iterator_specification is a *parallel iterator*. Otherwise, it is a *forward iterator*. Forward and reverse iterators are collectively called *sequential* iterators. In a reverse generalized iterator, the *iterator*_name shall be of a reversible iterator type. In a parallel generalized iterator, the *iterator*_name shall be of a parallel iterator type. In a reverse container element iterator, the default iterator type for the type of the *iterable*_name shall be a reversible iterator type. In a parallel container element iterator, the default iterator type for the type of the *iterable*_name shall be of a parallel iterator type. 4/5

{*AI05-0139-2*} {*AI12-0151-1*} {*AI12-0156-1*} {*AI12-0183-1*} The subtype defined by the loop_parameter_subtype_indication, if any, of a generalized iterator shall statically match the iteration cursor subtype. The subtype defined by the loop_parameter_subtype_indication, if any, of an array component iterator shall statically match the component subtype of the type of the *iterable*_name. The subtype defined by the loop_parameter_subtype_indication, if any, of a container element iterator shall statically match the default element subtype for the type of the *iterable*_name. 5/5

{*AI05-0139-2*} In a container element iterator whose *iterable*_name has type *T*, if the *iterable*_name denotes a constant or the Variable_Indexing aspect is not specified for *T*, then the Constant_Indexing aspect shall be specified for *T*. 6/3

{*AI12-0047-1*} The *iterator*_name or *iterable*_name of an iterator_specification shall not denote a subcomponent that depends on discriminants of an object whose nominal subtype is unconstrained, unless the object is known to be constrained. 6.1/4

Reason: This is the same rule that applies to renames; it serves the same purpose of preventing the object from disappearing while the iterator is still using it. 6.a/4

6.2/4 {*AI12-0120-1*} A container element iterator is illegal if the call of the default iterator function that creates the loop iterator (see below) is illegal.

6.b/4 **Ramification:** This can happen if the parameter to the default iterator function is **in out** and the *iterable*_name is a constant. The wording applies to any reason that the call would be illegal, as it's possible that one of the default parameters would be illegal, or that some accessibility check would fail.

6.3/4 {*AI12-0120-1*} A generalized iterator is illegal if the iteration cursor subtype of the *iterator*_name is a limited type at the point of the generalized iterator. A container element iterator is illegal if the default cursor subtype of the type of the *iterable*_name is a limited type at the point of the container element iterator.

6.c/4 **Reason:** If the cursor type is limited, the assignment to the loop parameter for a generalized iterator would be illegal. The same is true for a container element iterator. We have to say "at the point of the iterator" as the limitedness of a type can change due to visibility.

Static Semantics

7/5 {*AI05-0139-2*} {*AI05-0269-1*} {*AI05-0292-1*} {*AI12-0156-1*} An iterator_specification declares a *loop parameter*. In a generalized iterator, an array component iterator, or a container element iterator, if a loop_parameter_subtype_indication is present, it determines the nominal subtype of the loop parameter. In a generalized iterator, if a loop_parameter_subtype_indication is not present, the nominal subtype of the loop parameter is the iteration cursor subtype. In an array component iterator, if a loop_parameter_subtype_indication is not present, the nominal subtype of the loop parameter is the component subtype of the type of the *iterable*_name. In a container element iterator, if a loop_parameter_subtype_indication is not present, the nominal subtype of the loop parameter is the default element subtype for the type of the *iterable*_name.

8/3 {*AI05-0139-2*} {*AI05-0292-1*} In a generalized iterator, the loop parameter is a constant. In an array component iterator, the loop parameter is a constant if the *iterable*_name denotes a constant; otherwise it denotes a variable. In a container element iterator, the loop parameter is a constant if the *iterable*_name denotes a constant, or if the Variable_Indexing aspect is not specified for the type of the *iterable*_name; otherwise it is a variable.

8.a/5 **Ramification:** {*AI12-0093-1*} {*AI12-0156-1*} The loop parameter of a generalized iterator has the same accessibility as the loop statement. This means that the loop parameter object is finalized when the loop statement is left. (It also may be finalized as part of assigning a new value to the loop parameter.) For array component iterators, the loop parameter directly denotes an element of the array and has the accessibility of the associated array. For container element iterators, the loop parameter denotes the result of the indexing function call (in the case of a constant indexing) or a generalized reference thereof (in the case of a variable indexing). Roughly speaking, the loop parameter has the accessibility level of a single iteration of the loop. More precisely, the function result (or the generalized reference thereof) is considered to be renamed in the declarative part of a notional block statement which immediately encloses the loop's sequence_of_statements; the accessibility of the loop parameter is that of the block statement.

Dynamic Semantics

9/3 {*AI05-0139-2*} For the execution of a loop_statement with an iterator_specification, the iterator_specification is first elaborated. This elaboration elaborates the subtype_indication, if any.

10/5 {*AI05-0139-2*} {*AI12-0250-1*} {*AI12-0266-1*} For a sequential generalized iterator, the loop parameter is created, the *iterator*_name is evaluated, and the denoted iterator object becomes the *loop iterator*. In a forward generalized iterator, the operation First of the iterator type is called on the loop iterator, to produce the initial value for the loop parameter. If the result of calling Has_Element on the initial value is False, then the execution of the loop_statement is complete. Otherwise, the sequence_of_statements is conditionally executed and then the Next operation of the iterator type is called with the loop iterator and the current value of the loop parameter to produce the next value to be assigned to the loop parameter. This repeats until the result of calling Has_Element on the loop parameter is False, or the loop is left as a

consequence of a transfer of control. For a reverse generalized iterator, the operations Last and Previous are called rather than First and Next.

Ramification: {*AI12-0093-1*} The loop parameter of a generalized iterator is a variable of which the user only has a constant view. It follows the normal rules for a variable of its nominal subtype. In particular, if the nominal subtype is indefinite, the variable is constrained by its initial value. Similarly, if the nominal subtype is class-wide, the variable (like all variables) has the tag of the initial value. Constraint_Error may be raised by a subsequent iteration if Next or Previous return an object with a different tag or constraint. 10.a/4

{*AI12-0266-1*} For a parallel generalized iterator, the chunk_specification, if any, of the associated parallel construct, is first elaborated, to determine the maximum number of chunks (see 5.5), and then the operation Split_Into_Chunks of the iterator type is called, with the determined maximum passed as the Max_Chunks parameter, specifying the upper bound for the number of loop parameter objects (and the number of logical threads of control) to be associated with the iterator. In the absence of a chunk_specification, the maximum number of chunks is determined in an implementation-defined manner. 10.1/5

Implementation defined: The maximum number of chunks for a parallel generalized iterator without a chunk_specification. 10.b/5

Discussion: The Max_Chunks parameter of the Split_Into_Chunks procedure is an upper bound for the number of chunks to be associated with a loop. A container implementation may opt for a lower value for the number of chunks if a more optimal split can be determined. For instance, a tree-based container might create the split based on the number of branches at the top levels of the tree. 10.c/5

{*AI12-0250-1*} {*AI12-0266-1*} {*AI12-0418-1*} Upon return from Split_Into_Chunks, the actual number of chunks for the loop is determined by calling the Chunk_Count operation of the iterator, at which point one logical thread of control is initiated for each chunk, with an associated chunk index in the range from one to the actual number of chunks. 10.2/5

{*AI12-0250-1*} {*AI12-0266-1*} {*AI12-0418-1*} Within each logical thread of control, a loop parameter is created. If a chunk_specification with a discrete_subtype_definition is present in the associated parallel construct, then a chunk parameter is created and initialized with a value from the discrete subtype defined by the discrete_subtype_definition, so that the order of the chosen chunk parameter values correspond to the order of the chunk indices associated with the logical threads of control. The operation First of the iterator type that has a Chunk parameter is called on the loop iterator, with Chunk initialized from the corresponding chunk index, to produce the initial value for the loop parameter. If the result of calling Has_Element on this initial value is False, then the execution of the logical thread of control is complete. Otherwise, the sequence_of_statements is conditionally executed, and then the Next operation of the iterator type that has a Chunk parameter is called with the loop iterator, the current value of the loop parameter, and the corresponding chunk index, to produce the next value to be assigned to the loop parameter. This repeats until the result of calling Has_Element on the loop parameter is False, or the associated parallel construct is left as a consequence of a transfer of control. 10.3/5

{*AI12-0250-1*} {*AI12-0266-1*} {*AI12-0418-1*} In the absence of a transfer of control, the associated parallel construct of a parallel generalized iterator is complete when all of its logical threads of control are complete. 10.4/5

{*AI05-0139-2*} {*AI05-0292-1*} {*AI12-0250-1*} {*AI12-0266-1*} For an array component iterator, the chunk_specification of the associated parallel construct, if any, is first elaborated to determine the maximum number of chunks (see 5.5), and then the *iterable*_name is evaluated and the denoted array object becomes the *array for the loop*. If the array for the loop is a null array, then the execution of the loop_statement is complete. Otherwise, the sequence_of_statements is conditionally executed with the loop parameter denoting each component of the array for the loop, using a *canonical* order of components, which is last dimension varying fastest (unless the array has convention Fortran, in which case it is first dimension varying fastest). For a forward array component iterator, the iteration starts with 11/5

the component whose index values are each the first in their index range, and continues in the canonical order. For a reverse array component iterator, the iteration starts with the component whose index values are each the last in their index range, and continues in the reverse of the canonical order. For a parallel array component iterator, the iteration is broken up into contiguous chunks of the canonical order, such that all components are covered with no overlaps; each chunk has its own logical thread of control with its own loop parameter and iteration within each chunk is in the canonical order. The number of chunks is implementation defined, but is limited in the presence of a chunk_specification to the determined maximum. The loop iteration proceeds until the sequence_of_statements has been conditionally executed for each component of the array for the loop, or until the loop is left as a consequence of a transfer of control.

11.a/5 **Implementation defined:** The number of chunks for an array component iterator.

11.1/5 {*AI12-0266-1*} If a chunk_specification with a discrete_subtype_definition is present in the associated parallel construct, then the logical thread of control associated with a given chunk has a chunk parameter initialized with a distinct value from the discrete subtype defined by the discrete_subtype_definition. The values of the chunk parameters are assigned such that they increase in the canonical order of the starting array components for the chunks.

12/5 {*AI05-0139-2*} {*AI05-0292-1*} {*AI12-0111-1*} {*AI12-0266-1*} For a container element iterator, the chunk_specification of the associated parallel construct, if any, is first elaborated to determine the maximum number of chunks (see 5.5), and then the *iterable*_name is evaluated. If the container type has Iterator_View specified, an object of the Iterator_View type is created with the discriminant referencing the iterable container object denoted by the *iterable*_name. This is the *iterable container object for the loop*. Otherwise, the iterable container object denoted by the *iterable*_name becomes the iterable container object for the loop. The default iterator function for the type of the iterable container object for the loop is called on the iterable container object and the result is the *loop iterator*. For a sequential container element iterator, an object of the default cursor subtype is created (the *loop cursor*). For a parallel container element iterator, each chunk of iterations will have its own loop cursor, again of the default cursor subtype.

12.a/5 **Reason:** {*AI12-0111-1*} If Iterator_View is specified, we add an extra object and use that object for this iteration. This allows these iterators to automatically use the stable view (defined in each of the language-defined containers) to do the iteration. That eliminates the need to set and clear the tampering with elements indication each time Reference is called; that eliminates substantial overhead as finalization is typically used to implement the tampering reset.

13/5 {*AI05-0139-2*} {*AI05-0292-1*} {*AI12-0266-1*} A container element iterator then proceeds as described above for a generalized iterator, except that each reference to a loop parameter is replaced by a reference to the corresponding loop cursor. For a container element iterator, the loop parameter for each iteration instead denotes an indexing (see 4.1.6) into the iterable container object for the loop, with the only parameter to the indexing being the value of the loop cursor for the given iteration. If the loop parameter is a constant (see above), then the indexing uses the default constant indexing function for the type of the iterable container object for the loop; otherwise it uses the default variable indexing function.

14/4 {*AI12-0120-1*} Any exception propagated by the execution of a generalized iterator or container element iterator is propagated by the immediately enclosing loop statement.

14.a/4 **Ramification:** This text covers exceptions raised by called functions that make up the execution of the iterator as well as exceptions raised by the assignment to the loop parameter or cursor.

Examples

15/5 {*AI12-0429-1*} *Example of a parallel generalized loop over an array:*

16/5
```
{AI05-0269-1} {AI12-0266-1} {AI12-0429-1} parallel
for Element of Board loop  -- See 3.6.1.
   Element := Element * 2.0; -- Double each element of Board, a two-dimensional array.
end loop;
```

{AI05-0268-1} {AI12-0429-1} For examples of use of generalized iterators, see A.18.33 and the corresponding container packages in A.18.2 and A.18.3. 17/5

Extensions to Ada 2005

{AI05-0139-2} Generalized forms of loop iteration are new. 17.a/3

Incompatibilities With Ada 2012

{AI12-0047-1} **Corrigendum:** Added a rule to ensure that the object being iterated cannot be a component that could disappear before the loop completes. This could be incompatible by making a loop that was legal (and worked correctly, so long as the enclosing object is not modified during the loop) from the original Ada 2012 illegal in corrected Ada 2012. Such loops should be pretty rare, especially as these iterator forms are new to Ada 2012. 17.b/4

{AI12-0120-1} **Corrigendum:** Added rules to reject loops if the call to the default iterator function for a container element iterator is illegal, or if the cursor type of an iterator is limited. These are formally incompatible with original Ada 2012, but as it's unlikely that any Ada 2012 compiler ever allowed the illegal usages in an expansion of a loop (it's much more likely that they would have just caused an internal error in the compiler), this should have no effect in practice. 17.c/4

{AI12-0151-1} **Corrigendum:** Added a requirement that the given subtype statically match the subtype of the element or component for a component element iterator or array component iterator. Original Ada 2012 text allowed any type that covered the subtype of the element or component, but that led to questions of what the meaning was if they are different. In this case, the element is essentially a renaming of the container element, and it doesn't make sense for the constraints to be different. Ignoring explicitly defined constraints in renames is a mistake that we don't want to continue, thus we require static matching. This means that some programs might be illegal, but those programs were misleading at best, and potentially would raise unexpected exceptions because the element values might have been invalid or abnormal with respect to the declared constraint. 17.d/4

Extensions to Ada 2012

{AI12-0156-1} For consistency, we now allow a subtype_indication on a generalized iterator, and anonymous access types on all forms of iterator. We introduced a new syntax non-terminal, loop_parameter_subtype_indication to simplfy the wording. 17.e/5

{AI12-0250-1} An iterator_filter is now allowed on iterator_specifications. This is mainly for consistency with aggregate and reduction iterators, where it eliminates the need for temporary objects. 17.f/5

Wording Changes from Ada 2012

{AI12-0120-1} **Corrigendum:** Added wording to specify that a loop propagates any exceptions propagated by the execution of an iterator. Since that's what naturally would happen from a macro-style expansion of the parts of an iterator, and no other interpretation makes sense given the way the rest of Ada works, we consider it so unlikely that any Ada 2012 implementation ever did anything else that we don't document this as a possible inconsistency. 17.g/4

{AI12-0111-1} Added wording to include the use of the iterator view in a container element iterator. 17.h/5

{AI12-0266-1} Added wording to describe the execution of parallel iterators. 17.i/5

5.5.3 Procedural Iterators

{AI12-0189-1} A procedural_iterator invokes a user-defined procedure, passing in the body of the enclosing loop_statement as a parameter of an anonymous access-to-procedure type, to allow the loop body to be executed repeatedly as part of the invocation of the user-defined procedure. 1/5

Syntax

{AI12-0189-1} {AI12-0250-1} procedural_iterator ::= 2/5
    iterator_parameter_specification **of** iterator_procedure_call
      [iterator_filter]

{AI12-0189-1} {AI12-0308-1} iterator_parameter_specification ::= 3/5
    formal_part
  | (defining_identifier{, defining_identifier})

4/5 {*AI12-0189-1*} iterator_procedure_call ::=
*procedure*_name
| *procedure*_prefix iterator_actual_parameter_part

5/5 {*AI12-0189-1*} iterator_actual_parameter_part ::=
(iterator_parameter_association {, iterator_parameter_association})

6/5 {*AI12-0189-1*} iterator_parameter_association ::=
parameter_association
| parameter_association_with_box

7/5 {*AI12-0189-1*} parameter_association_with_box ::=
[*formal_parameter*_selector_name =>] <>

8/5 {*AI12-0189-1*} At most one iterator_parameter_association within an iterator_actual_parameter_part shall be a parameter_association_with_box.

Name Resolution Rules

9/5 {*AI12-0189-1*} {*AI12-0292-1*} {*AI12-0326-2*} The name or prefix given in an iterator_procedure_call shall resolve to denote a callable entity *C* (the *iterating procedure*) that is a procedure, or an entry renamed as (viewed as) a procedure. [When there is an iterator_actual_parameter_part, the prefix can be an implicit_dereference of an access-to-subprogram value.]

10/5 {*AI12-0189-1*} An iterator_procedure_call without a parameter_association_with_box is equivalent to one with an iterator_actual_parameter_part with an additional parameter_association_with_box at the end, with the *formal_parameter*_selector_name identifying the last formal parameter of the callable entity denoted by the name or prefix.

11/5 {*AI12-0189-1*} {*AI12-0320-1*} An iterator_procedure_call shall contain at most one iterator_parameter_association for each formal parameter of the callable entity *C*. Each formal parameter without an iterator_parameter_association shall have a default_expression (in the profile of the view of *C* denoted by the name or prefix).

12/5 {*AI12-0189-1*} The formal parameter of the callable entity *C* associated with the parameter_association_with_box shall be of an anonymous access-to-procedure type *A*.

Legality Rules

13/5 {*AI12-0189-1*} {*AI12-0308-1*} The anonymous access-to-procedure type *A* shall have at least one formal parameter in its parameter profile. If the iterator_parameter_specification is a formal_part, then this formal_part shall be mode conformant with that of *A*. If the iterator_parameter_specification is a list of defining_identifiers, the number of formal parameters of *A* shall be the same as the length of this list.

14/5 {*AI12-0189-1*} {*AI12-0292-1*} [If the name or prefix given in an iterator_procedure_call denotes an abstract subprogram, the subprogram shall be a dispatching subprogram.]

14.a/5 **Proof:** {*AI12-0320-1*} This is stated normatively in 3.9.3.

Static Semantics

15/5 {*AI12-0189-1*} {*AI12-0250-1*} {*AI12-0308-1*} {*AI12-0326-2*} A loop_statement with an iteration_scheme that has a procedural_iterator is equivalent to a local declaration of a procedure P followed by a procedure_call_statement that is formed from the iterator_procedure_call by replacing the <> of the parameter_association_with_box with P'Access. The formal_part of the locally declared procedure P is formed from the formal_part of the anonymous access-to-procedure type *A*, by replacing the identifier of each formal parameter of this formal_part with the identifier of the corresponding formal

parameter or element of the list of defining_identifiers given in the iterator_parameter_specification. The body of *P* consists of the conditionally executed sequence_of_statements. The procedure P is called the *loop body procedure*.

Implementation Note: {*AI12-0250-1*} For a procedural_iterator with an iterator_filter, the body of the routine would be something like: 15.a/5

```
procedure P ... is
begin
   if iterator_filter then
      sequence_of_statements
   end if;
end P;
```
15.b/5

{*AI12-0344-1*} In a procedural iterator, the Parallel_Calls aspect (see 9.10.1) of the loop body procedure is True if the reserved word **parallel** occurs in the corresponding loop statement, and False otherwise. 16/5

{*AI12-0189-1*} {*AI12-0326-2*} {*AI12-0344-1*} The following aspects may be specified for a callable entity *S* that has exactly one formal parameter of an anonymous access-to-subprogram type: 17/5

Allows_Exit 18/5

The Allows_Exit aspect is of type Boolean. The specified value shall be static. The Allows_Exit aspect of an inherited primitive subprogram is True if Allows_Exit is True either for the corresponding subprogram of the progenitor type or for any other inherited subprogram that it overrides. If not specified or inherited as True, the Allows_Exit aspect of a callable entity is False. For an entry, only a confirming specification of False is permitted for the Allows_Exit aspect.

Reason: An entry does not allow exit, because implementing a transfer of control out of a task or protected entry creates unnecessarily complex dynamic semantics. 18.a/5

Specifying the Allows_Exit aspect to be True for a subprogram indicates that the subprogram *allows exit*, meaning that it is prepared to be completed by arbitrary transfers of control from the loop body procedure[, including propagation of exceptions. A subprogram for which Allows_Exit is True should use finalization as appropriate rather than exception handling to recover resources and make any necessary final updates to data structures]. 19/5

Aspect Description for Allows_Exit: An indication of whether a subprogram will operate correctly for arbitrary transfers of control. 19.a/5

Ramification: A subprogram that does not need cleanup satisfies the requirements, and thus can specify Allows_Exit as True. If a subprogram *S* allows exit, it cannot expect to get control other than via finalization if the loop body procedure initiates a transfer of control as part of a procedural_iterator. In particular, exception handlers in *S*, even **when others** handlers, will not be executed when a transfer of control occurs. The mechanism that the implementation uses to implement such transfers of control needs to avoid triggering exception handlers. 19.b/5

Parallel_Iterator 20/5

The Parallel_Iterator aspect is of type Boolean. The specified value shall be static. The Parallel_Iterator aspect of an inherited primitive subprogram is True if Parallel_Iterator is True either for the corresponding subprogram of the progenitor type or for any other inherited subprogram that it overrides. If not specified or inherited as True, the Parallel_Iterator aspect of a callable entity is False.

{*AI12-0189-1*} {*AI12-0442-1*} Specifying the Parallel_Iterator aspect to be True for a callable entity indicates that the entity is allowed to invoke the loop body procedure from multiple distinct logical threads of control. The Parallel_Iterator aspect for a subprogram shall be statically False if the subprogram allows exit. 21/5

Aspect Description for Parallel_Iterator: An indication of whether a subprogram may use multiple threads of control to invoke a loop body procedure. 21.a/5

Reason: Permitting exit from a parallel procedural iterator introduces additional semantic and implementation complexity. 21.b/5

Legality Rules

22/5 {*AI12-0189-1*} {*AI12-0326-2*} If a callable entity overrides an inherited dispatching subprogram that allows exit, the overriding callable entity also shall allow exit. If a callable entity overrides an inherited dispatching subprogram that has a True Parallel_Iterator aspect, the overriding callable entity also shall have a True Parallel_Iterator aspect.

22.a/5 **Ramification:** Since an entry never allows exit, attempting to implement an allows exit subprogram with a task or protected entry is always illegal. However, the Parallel_Iterator aspect can be applied to an entry, so a subprogram with the Parallel_Iterator aspect True can be implemented by an entry.

23/5 {*AI12-0326-2*} A loop_statement with a procedural_iterator as its iteration_scheme shall begin with the reserved word **parallel** if and only if the callable entity identified in the iterator_procedure_call has a Parallel_iterator aspect of True.

24/5 {*AI12-0326-2*} If the actual parameter of an anonymous access-to-subprogram type, passed in an explicit call of a subprogram for which the Parallel_Iterator aspect is True, is of the form *P*'Access, the designated subprogram *P* shall have a Parallel_Calls aspect True (see 9.10.1).

25/5 {*AI12-0189-1*} {*AI12-0326-2*} The sequence_of_statements of a loop_statement with a procedural_iterator as its iteration_scheme shall contain an exit_statement, return statement, goto_statement, or requeue_statement that leaves the loop only if the callable entity associated with the procedural_iterator allows exit.

26/5 {*AI12-0294-1*} The sequence_of_statements of a loop_statement with a procedural_iterator as its iteration_scheme shall not contain an accept_statement whose entry_declaration occurs outside the loop_statement.

26.a/5 **Reason:** An accept_statement is not allowed in a procedure (see 9.5.2), it has to be directly in a task_body. Since the loop body here is implemented as a procedure, we can't allow accept_statements there, either, even if the loop itself is directly in a task_body.

26.b/5 **Ramification:** This includes cases where the accept_statement is part of another construct, for instance, a select_statement.

Dynamic Semantics

27/5 {*AI12-0326-2*} {*AI12-0355-2*} {*AI12-0445-1*} [For the execution of a loop_statement with an iteration_scheme that has a procedural_iterator, the procedure denoted by the name or prefix of the iterator_procedure_call (the *iterating procedure*) is invoked, passing an access value designating the loop body procedure as a parameter. The iterating procedure then calls the loop body procedure zero or more times and returns, whereupon the loop_statement is complete. If the **parallel** reserved word is present, the iterating procedure is allowed to invoke the loop body procedure from multiple distinct logical threads of control.] The aspect_specification, if any, is elaborated prior to the invocation of the iterating procedure.

27.a/5 **Proof:** The stated dynamic semantics are implied by the static semantics given above and the bounded errors given below.

Bounded (Run-Time) Errors

28/5 {*AI12-0326-2*} {*AI12-0445-1*} If the callable entity identified in the iterator_procedure_call allows exit, then it is a bounded error for a call of the loop body procedure to be performed from within an abort-deferred operation (see 9.8), unless the entire loop_statement was within the same abort-deferred operation. If detected, Program_Error is raised at the point of the call; otherwise, a transfer of control from the sequence_of_statements of the loop_statement will not necessarily terminate the loop_statement, and the loop body procedure can be called again.

{*AI12-0326-2*} {*AI12-0445-1*} If a loop_statement with the procedural_iterator as its iteration_scheme (see 5.5) does not begin with the reserved word **parallel**, it is a bounded error if the loop body procedure is invoked from a different logical thread of control than the one that initiates the loop_statement. If detected, Program_Error is raised; otherwise, conflicts associated with concurrent executions of the loop body procedure can occur without being detected by the applicable conflict check policy (see 9.10.1). Furthermore, propagating an exception or making an attempt to exit in the presence of multiple threads of control will not necessarily terminate the loop_statement, deadlock can occur, or the loop body procedure can be called again. 29/5

Discussion: {*AI12-0326-2*} Other Ada rules are still in effect for the allows exit subprogram *A*, of course. For instance, if a transfer of control causes finalization which raises an exception, Program_Error will be propagated by *A* (rather than the transfer of control). In such a case, the bounded error above would still apply. Another example is the case where an unrelated task is waiting on the normal completion of the loop body procedure call in *A*. Such a task might end up waiting forever if a transfer of control happens (this is a deadlock situation). This case does not require additional wording, as the same thing would happen if an exception is propagated from the loop body procedure or if *A* executed a transfer of control (such as a return statement). 29.a/5

Examples

{*AI12-0189-1*} {*AI12-0379-1*} {*AI12-0429-1*} *Example of iterating over a map from My_Key_Type to My_Element_Type (see A.18.4):* 30/5

```
for (C : Cursor) of My_Map.Iterate loop                                    31/5
   Put_Line (My_Key_Type'Image (Key (C)) & " => " &
      My_Element_Type'Image (Element (C)));
end loop;
```

-- The above is equivalent to: 32/5

```
declare                                                                    33/5
   procedure P (C : Cursor) is
   begin
      Put_Line (My_Key_Type'Image (Key (c)) & " => " &
         My_Element_Type'Image (Element (C)));
   end P;
begin
   My_Map.Iterate (P'Access);
end;
```

{*AI12-0189-1*} {*AI12-0429-1*} *Example of iterating over the environment variables (see A.17):* 34/5

```
for (Name, Val) of Ada.Environment_Variables.Iterate(<>) loop              35/5
   -- "(<>)" is optional because it is the last parameter
   Put_Line (Name & " => " & Val);
end loop;
```

-- The above is equivalent to: 36/5

```
declare                                                                    37/5
   procedure P (Name : String; Val : String) is
   begin
      Put_Line (Name & " => " & Val);
   end P;
begin
   Ada.Environment_Variables.Iterate (P'Access);
end;
```

Extensions to Ada 2012

{*AI12-0189-1*} {*AI12-0292-1*} {*AI12-0294-1*} {*AI12-0326-2*} {*AI12-0344-1*} Procedural iterators, and the Allows_Exit and Parallel_Iterator aspects are new in Ada 2022. 37.a/5

5.6 Block Statements

[A block_statement encloses a handled_sequence_of_statements optionally preceded by a declarative_part.] 1

Syntax

2 block_statement ::=
  [*block*_statement_identifier:]
    [**declare**
      declarative_part]
    begin
      handled_sequence_of_statements
    end [*block*_identifier];

3 If a block_statement has a *block*_statement_identifier, then the identifier shall be repeated after the **end**; otherwise, there shall not be an identifier after the **end**.

Static Semantics

4 A block_statement that has no explicit declarative_part has an implicit empty declarative_part.

4.a **Ramification:** Thus, other rules can always refer to the declarative_part of a block_statement.

Dynamic Semantics

5 The execution of a block_statement consists of the elaboration of its declarative_part followed by the execution of its handled_sequence_of_statements.

Examples

6 *Example of a block statement with a local variable:*

7
```
Swap:
   declare
      Temp : Integer;
   begin
      Temp := V; V := U; U := Temp;
   end Swap;
```

7.a **Ramification:** If task objects are declared within a block_statement whose execution is completed, the block_statement is not left until all its dependent tasks are terminated (see 7.6). This rule applies to completion caused by a transfer of control.

7.b Within a block_statement, the block name can be used in expanded names denoting local entities such as Swap.Temp in the above example (see 4.1.3).

Wording Changes from Ada 83

7.c The syntax rule for block_statement now uses the syntactic category handled_sequence_of_statements.

5.6.1 Parallel Block Statements

1/5 {*AI12-0119-1*} {*AI12-0436-1*} [A parallel_block_statement comprises two or more sequence_of_statements separated by **and** where each represents an independent activity that is intended to proceed concurrently with the others.]

Syntax

2/5 {*AI12-0119-1*} {*AI12-0427-1*} {*AI12-0436-1*} parallel_block_statement ::=
  parallel [(chunk_specification)] [aspect_specification] **do**
    sequence_of_statements
  and
    sequence_of_statements
  {**and**
    sequence_of_statements}

end do;

{*AI12-0427-1*} The chunk_specification, if any, of a parallel_block_statement shall be an *integer*_simple_expression. 3/5

Dynamic Semantics

{*AI12-0119-1*} {*AI12-0427-1*} For the execution of a parallel_block_statement, the chunk_specification and the aspect_specification, if any, are elaborated in an arbitrary order. After elaborating the chunk_specification, if any, a check is made that the determined maximum number of chunks is greater than zero. If this check fails, Program_Error is raised. 4/5

{*AI12-0119-1*} {*AI12-0427-1*} {*AI12-0436-1*} {*AI12-0449-1*} Then, the various sequence_of_statements are grouped into one or more *chunks*, each with its own logical thread of control (see Clause 9), up to the maximum number of chunks specified by the chunk_specification, if any. Within each chunk every sequence_of_statements of the chunk is executed in turn, in an arbitrary order. The parallel_block_statement is complete once every one of the sequence_of_statements has completed, either by reaching the end of its execution, or due to a transfer of control out of the construct by one of the sequence_of_statements (see 5.1). 5/5

Implementation Note: {*AI12-0119-1*} {*AI12-0436-1*} Although each sequence_of_statements of a parallel block represents a separate logical thread of control, the implementation may choose to combine two or more such logical threads of control into a single physical thread of control to reduce the cost of creating numerous physical threads of control. 5.a/5

Examples

{*AI12-0429-1*} *Example of a parallel block used to walk a binary tree in parallel:* 6/5

```
{AI12-0119-1} {AI12-0404-1} procedure Traverse (T : Expr_Ptr) is -- see 3.9.1
begin
   if T /= null and then
      T.all in Binary_Operation'Class -- see 3.9.1
   then -- recurse down the binary tree
      parallel do
         Traverse (T.Left);
      and
         Traverse (T.Right);
      and
         Ada.Text_IO.Put_Line
            ("Processing " & Ada.Tags.Expanded_Name (T'Tag));
      end do;
   end if;
end Traverse;
```
7/5

8/5 *{AI12-0429-1} Example of a parallel block used to search two halves of a string in parallel:*

9/5
```
{AI12-0119-1} function Search (S : String; Char : Character) return Boolean is
begin
   if S'Length <= 1000 then
      -- Sequential scan
      return (for some C of S => C = Char);
   else
      -- Parallel divide and conquer
      declare
         Mid : constant Positive := S'First + S'Length/2 - 1;
      begin
         parallel do
            for C of S(S'First .. Mid) loop
               if C = Char then
                  return True;  -- Terminates enclosing do
               end if;
            end loop;
         and
            for C of S(Mid + 1 .. S'Last) loop
               if C = Char then
                  return True;  -- Terminates enclosing do
               end if;
            end loop;
         end do;
         -- Not found
         return False;
      end;
   end if;
end Search;
```

Extensions to Ada 2012

9.a/5 {*AI12-0119-1*} {*AI12-0427-1*} {*AI12-0436-1*} The parallel_block_statement is new.

5.7 Exit Statements

1 [An exit_statement is used to complete the execution of an enclosing loop_statement; the completion is conditional if the exit_statement includes a condition.]

Syntax

2 exit_statement ::=
exit [*loop_*name] [**when** condition];

Name Resolution Rules

3 The *loop_*name, if any, in an exit_statement shall resolve to denote a loop_statement.

Legality Rules

4 Each exit_statement *applies to* a loop_statement; this is the loop_statement being exited. An exit_-statement with a name is only allowed within the loop_statement denoted by the name, and applies to that loop_statement. An exit_statement without a name is only allowed within a loop_statement, and applies to the innermost enclosing one. An exit_statement that applies to a given loop_statement shall not appear within a body or accept_statement, if this construct is itself enclosed by the given loop_statement.

Dynamic Semantics

For the execution of an exit_statement, the condition, if present, is first evaluated. If the value of the condition is True, or if there is no condition, a transfer of control is done to complete the loop_statement. If the value of the condition is False, no transfer of control takes place. 5

NOTE Several nested loops can be exited by an exit_statement that names the outer loop. 6

Examples

Examples of loops with exit statements: 7

```
for N in 1 .. Max_Num_Items loop                                    8
   Get_New_Item(New_Item);
   Merge_Item(New_Item, Storage_File);
   exit when New_Item = Terminal_Item;
end loop;

Main_Cycle:                                                         9
   loop
      -- initial statements
      exit Main_Cycle when Found;
      -- final statements
   end loop Main_Cycle;
```

5.8 Goto Statements

[A goto_statement specifies an explicit transfer of control from this statement to a target statement with a given label.] 1

Syntax

goto_statement ::= **goto** *label_*name; 2

Name Resolution Rules

The *label_*name shall resolve to denote a label; the statement with that label is the *target statement*. 3

Legality Rules

The innermost sequence_of_statements that encloses the target statement shall also enclose the goto_statement. Furthermore, if a goto_statement is enclosed by an accept_statement or a body, then the target statement shall not be outside this enclosing construct. 4

Ramification: The goto_statement can be a statement of an inner sequence_. 4.a

It follows from the second rule that if the target statement is enclosed by such a construct, then the goto_statement cannot be outside. 4.b

Dynamic Semantics

The execution of a goto_statement transfers control to the target statement, completing the execution of any compound_statement that encloses the goto_statement but does not enclose the target. 5

NOTE The above rules allow transfer of control to a statement of an enclosing sequence_of_statements but not the reverse. Similarly, they prohibit transfers of control such as between alternatives of a case_statement, if_statement, or select_statement; between exception_handlers; or from an exception_handler of a handled_sequence_of_statements back to its sequence_of_statements. 6

Examples

7 *Example of a loop containing a goto statement:*

8
```
<<Sort>>
for I in 1 .. N-1 loop
   if A(I) > A(I+1) then
      Exchange(A(I), A(I+1));
      goto Sort;
   end if;
end loop;
```

6 Subprograms

A subprogram is a program unit or intrinsic operation whose execution is invoked by a subprogram call. There are two forms of subprogram: procedures and functions. A procedure call is a statement; a function call is an expression and returns a value. The definition of a subprogram can be given in two parts: a subprogram declaration defining its interface, and a subprogram_body defining its execution. [Operators and enumeration literals are functions.] 1

To be honest: A function call is an expression, but more specifically it is a name. 1.a

Term entry: subprogram — unit of a program that can be brought into execution in various contexts, with the invocation being a subprogram call that can parameterize the effect of the subprogram through the passing of operands Note: There are two forms of subprograms: functions, which return values, and procedures, which do not. 1.b/5

Term entry: function — form of subprogram that returns a result and can be called as part of an expression 1.c/5

Term entry: procedure — form of subprogram that does not return a result and can only be invoked by a statement 1.d/5

{*AI05-0299-1*} A *callable entity* is a subprogram or entry (see Section 9). A callable entity is invoked by a *call*; that is, a subprogram call or entry call. A *callable construct* is a construct that defines the action of a call upon a callable entity: a subprogram_body, entry_body, or accept_statement. 2/3

Ramification: Note that "callable entity" includes predefined operators, enumeration literals, and abstract subprograms. "Call" includes calls of these things. They do not have callable constructs, since they don't have completions. 2.a

6.1 Subprogram Declarations

[A subprogram_declaration declares a procedure or function.] 1

Syntax

{*AI95-00218-03*} {*AI05-0183-1*} subprogram_declaration ::= 2/3
  [overriding_indicator]
  subprogram_specification
    [aspect_specification];

This paragraph was deleted.{*AI95-00348-01*} 3/2

{*AI95-00348-01*} subprogram_specification ::= 4/2
  procedure_specification
 | function_specification

{*AI95-00348-01*} procedure_specification ::= 4.1/2
  procedure defining_program_unit_name parameter_profile

{*AI95-00348-01*} function_specification ::= 4.2/2
  function defining_designator parameter_and_result_profile

designator ::= [parent_unit_name .]identifier | operator_symbol 5

defining_designator ::= 6
  defining_program_unit_name | defining_operator_symbol

defining_program_unit_name ::= [parent_unit_name .]defining_identifier 7

[The optional parent_unit_name is only allowed for library units (see 10.1.1).] 8

operator_symbol ::= string_literal 9

10/5 {*AI95-00395-01*} {*AI05-0299-1*} {*AI12-0449-1*} The sequence of characters in an operator_symbol shall form a reserved word, a delimiter, or compound delimiter that corresponds to an operator belonging to one of the six categories of operators defined in 4.5.

10.a/3 **Reason:** {*AI95-00395-01*} {*AI05-0090-1*} The "sequence of characters" of the string literal of the operator is a technical term (see 2.6), and does not include the surrounding quote characters. As defined in 2.2, lexical elements are "formed" from a sequence of characters. Spaces are not allowed, and upper and lower case is not significant.

11 defining_operator_symbol ::= operator_symbol

12 parameter_profile ::= [formal_part]

13/2 {*AI95-00231-01*} {*AI95-00318-02*} parameter_and_result_profile ::=
[formal_part] **return** [null_exclusion] subtype_mark
| [formal_part] **return** access_definition

14 formal_part ::=
(parameter_specification {; parameter_specification})

15/5 {*AI95-00231-01*} {*AI05-0142-4*} {*AI12-0395-1*} parameter_specification ::=
defining_identifier_list : [**aliased**] mode [null_exclusion] subtype_mark [:= default_expression]
[aspect_specification]
| defining_identifier_list : access_definition [:= default_expression]
[aspect_specification]

15.a/5 **Discussion:** {*AI12-0395-1*} Only implementation-defined aspects are allowed on formal parameters in Ada 2022. Implementers are cautioned that any aspect allowed on a formal parameter will need conformance rules. If, for instance, an aspect changed the representation of a parameter, rules would be needed to ensure that the representation is the same for the specification and body.

16 mode ::= [**in**] | **in out** | **out**

Name Resolution Rules

17 A *formal parameter* is an object [directly visible within a subprogram_body] that represents the actual parameter passed to the subprogram in a call; it is declared by a parameter_specification. For a formal parameter, the expected type for its default_expression, if any, is that of the formal parameter.

Legality Rules

18/3 {*AI05-0143-1*} The *parameter mode* of a formal parameter conveys the direction of information transfer with the actual parameter: **in**, **in out**, or **out**. Mode **in** is the default, and is the mode of a parameter defined by an access_definition.

18.a/3 *This paragraph was deleted.*{*AI05-0143-1*}

19 A default_expression is only allowed in a parameter_specification for a formal parameter of mode **in**.

20/3 {*AI95-00348-01*} {*AI05-0177-1*} {*AI05-0229-1*} A subprogram_declaration or a generic_subprogram_declaration requires a completion [unless the Import aspect (see B.1) is True for the declaration; the completion shall be a body or a renaming_declaration (see 8.5)]. [A completion is not allowed for an abstract_subprogram_declaration (see 3.9.3), a null_procedure_declaration (see 6.7), or an expression_function_declaration (see 6.8).]

20.a/3 **Ramification:** {*AI95-00348-01*} {*AI05-0177-1*} Abstract subprograms , null procedures, and expression functions are not declared by subprogram_declarations, and so do not require completion (although the latter two can *be* completions). Protected subprograms are declared by subprogram_declarations, and so require completion. Note that an abstract subprogram is a subprogram, a null procedure is a subprogram, an expression function is a subprogram, and a protected subprogram is a subprogram, but a generic subprogram is not a subprogram.

20.b/3 **Proof:** {*AI05-0229-1*} When the Import aspect is True for any entity, no completion is allowed (see B.1).

A name that denotes a formal parameter is not allowed within the formal_part in which it is declared, nor within the formal_part of a corresponding body or accept_statement. 21

Ramification: By contrast, generic_formal_parameter_declarations are visible to subsequent declarations in the same generic_formal_part. 21.a

Static Semantics

The *profile* of (a view of) a callable entity is either a parameter_profile or parameter_and_result_profile[; it embodies information about the interface to that entity — for example, the profile includes information about parameters passed to the callable entity. All callable entities have a profile — enumeration literals, other subprograms, and entries. An access-to-subprogram type has a designated profile.] Associated with a profile is a calling convention. A subprogram_declaration declares a procedure or a function, as indicated by the initial reserved word, with name and profile as given by its specification. 22

{*AI95-00231-01*} {*AI95-00318-02*} The nominal subtype of a formal parameter is the subtype determined by the optional null_exclusion and the subtype_mark, or defined by the access_definition, in the parameter_specification. The nominal subtype of a function result is the subtype determined by the optional null_exclusion and the subtype_mark, or defined by the access_definition, in the parameter_and_result_profile. 23/2

{*AI05-0142-4*} An *explicitly aliased parameter* is a formal parameter whose parameter_specification includes the reserved word **aliased**. 23.1/3

{*AI95-00231-01*} {*AI95-00254-01*} {*AI95-00318-02*} An *access parameter* is a formal **in** parameter specified by an access_definition. An *access result type* is a function result type specified by an access_definition. An access parameter or result type is of an anonymous access type (see 3.10). [Access parameters of an access-to-object type allow dispatching calls to be controlled by access values. Access parameters of an access-to-subprogram type permit calls to subprograms passed as parameters irrespective of their accessibility level.] 24/2

Discussion: {*AI95-00318-02*} Access result types have normal accessibility and thus don't have any special properties worth noting here. 24.a/2

The *subtypes of a profile* are: 25

- For any non-access parameters, the nominal subtype of the parameter. 26
- {*AI95-00254-01*} For any access parameters of an access-to-object type, the designated subtype of the parameter type. 27/2
- {*AI95-00254-01*} {*AI05-0164-1*} For any access parameters of an access-to-subprogram type, the subtypes of the designated profile of the parameter type. 27.1/3
- {*AI95-00231-01*} {*AI95-00318-02*} For any non-access result, the nominal subtype of the function result. 28/2
- {*AI95-00318-02*} For any access result type of an access-to-object type, the designated subtype of the result type. 28.1/2
- {*AI95-00318-02*} {*AI05-0164-1*} For any access result type of an access-to-subprogram type, the subtypes of the designated profile of the result type. 28.2/3

[The *types of a profile* are the types of those subtypes.] 29

{*AI95-00348-01*} {*AI05-0177-1*} [A subprogram declared by an abstract_subprogram_declaration is abstract; a subprogram declared by a subprogram_declaration is not. See 3.9.3, “Abstract Types and Subprograms”. Similarly, a procedure declared by a null_procedure_declaration is a null procedure; a procedure declared by a subprogram_declaration is not. See 6.7, “Null Procedures”. Finally, a function 30/3

declared by an expression_function_declaration is an expression function; a function declared by a subprogram_declaration is not. See 6.8, “Expression Functions”.]

30.1/2 {*AI95-00218-03*} [An overriding_indicator is used to indicate whether overriding is intended. See 8.3.1, “Overriding Indicators”.]

Dynamic Semantics

31/2 {*AI95-00348-01*} The elaboration of a subprogram_declaration has no effect.

32 NOTE 1 A parameter_specification with several identifiers is equivalent to a sequence of single parameter_specifications, as explained in 3.3.

33 NOTE 2 Abstract subprograms do not have bodies, and cannot be used in a nondispatching call (see 3.9.3, “Abstract Types and Subprograms”).

34 NOTE 3 The evaluation of default_expressions is caused by certain calls, as described in 6.4.1. They are not evaluated during the elaboration of the subprogram declaration.

35 NOTE 4 Subprograms can be called recursively and can be called concurrently from multiple tasks.

Examples

36 *Examples of subprogram declarations:*

37
```
procedure Traverse_Tree;
procedure Increment(X : in out Integer);
procedure Right_Indent(Margin : out Line_Size);          -- see 3.5.4
procedure Switch(From, To : in out Link);                -- see 3.10.1
```

38
```
function Random return Probability;                      -- see 3.5.7
```

39/4
```
{AI12-0056-1} function Min_Cell(X : Link) return Cell;                    -- see 3.10.1
function Next_Frame(K : Positive) return Frame;          -- see 3.10
function Dot_Product(Left, Right : Vector) return Real;  -- see 3.6
function Find(B : aliased in out Barrel; Key : String) return Real;
                                                         -- see 4.1.5
```

40
```
function "*"(Left, Right : Matrix) return Matrix;        -- see 3.6
```

41 *Examples of **in** parameters with default expressions:*

42
```
procedure Print_Header(Pages  : in Natural;
            Header : in Line    :=  (1 .. Line'Last => ' '); -- see 3.6
            Center : in Boolean := True);
```

Extensions to Ada 83

42.a The syntax for abstract_subprogram_declaration is added. The syntax for parameter_specification is revised to allow for access parameters (see 3.10)

42.b/3 {*AI05-0299-1*} Program units that are library units may have a parent_unit_name to indicate the parent of a child (see 10.1.1).

Wording Changes from Ada 83

42.c We have incorporated the rules from RM83-6.5, “Function Subprograms” here and in 6.3, “Subprogram Bodies”

42.d We have incorporated the definitions of RM83-6.6, “Parameter and Result Type Profile - Overloading of Subprograms” here.

42.e The syntax rule for defining_operator_symbol is new. It is used for the defining occurrence of an operator_symbol, analogously to defining_identifier. Usage occurrences use the direct_name or selector_name syntactic categories. The syntax rules for defining_designator and defining_program_unit_name are new.

Extensions to Ada 95

42.f/2 {*AI95-00218-03*} Subprograms now allow overriding_indicators for better error checking of overriding.

42.g/2 {*AI95-00231-01*} An optional null_exclusion can be used in a formal parameter declaration. Similarly, an optional null_exclusion can be used in a function result.

42.h/2 {*AI95-00318-02*} The return type of a function can be an anonymous access type.

Wording Changes from Ada 95

{*AI95-00254-01*} A description of the purpose of anonymous access-to-subprogram parameters and the definition of the profile of subprograms containing them was added. 42.i/2

{*AI95-00348-01*} Split the production for subprogram_specification in order to make the declaration of null procedures (see 6.7) easier. 42.j/2

{*AI95-00348-01*} Moved the Syntax and Dynamic Semantics for abstract_subprogram_declaration to 3.9.3, so that the syntax and semantics are together. This also keeps abstract and null subprograms similar. 42.k/2

{*AI95-00395-01*} Revised to allow other_format characters in operator_symbols in the same way as the underlying constructs. 42.l/2

Extensions to Ada 2005

{*AI05-0142-4*} Parameters can now be explicitly aliased, allowing parts of function results to designate parameters and forcing by-reference parameter passing. 42.m/3

{*AI05-0143-1*} The parameters of a function can now have any mode. 42.n/3

{*AI05-0183-1*} An optional aspect_specification can be used in a subprogram_declaration. This is described in 13.1.1. 42.o/3

Wording Changes from Ada 2005

{*AI05-0177-1*} Added expression functions (see 6.8) to the wording. 42.p/3

Extensions to Ada 2012

{*AI12-0395-1*} Parameters now can have an aspect_specification, allowing the specification of (implementation-defined) aspects for individual parameters. 42.q/5

6.1.1 Preconditions and Postconditions

{*AI05-0145-2*} {*AI05-0247-1*} {*AI12-0045-1*} {*AI12-0220-1*} {*AI12-0272-1*} {*AI12-0396-1*} For a noninstance subprogram [(including a generic formal subprogram)], a generic subprogram, an entry, or an access-to-subprogram type, the following language-defined assertion aspects may be specified with an aspect_specification (see 13.1.1): 1/5

Ramification: {*AI12-0045-1*} "Noninstance subprogram" excludes a subprogram that is an instance of a generic subprogram. In that case, the aspects should be specified on the generic subprogram. If preconditions or postconditions need to be added to an instance of a generic subprogram, it can be accomplished by creating a separate subprogram specification and then completing that specification with a renames-as-body of the instance. 1.a/4

Proof: {*AI12-0272-1*} A generic formal subprogram is a subprogram, and there are no rules to prevent using these attributes on it. 1.b/5

{*AI12-0220-1*} Pre 2/5

This aspect specifies a specific precondition for a callable entity or an access-to-subprogram type; it shall be specified by an expression, called a *specific precondition expression*. If not specified for an entity, the specific precondition expression for the entity is the enumeration literal True.

To be honest: In this and the following rules, we are talking about the enumeration literal True declared in package Standard (see A.1), and not some other value or identifier True. That matters as some rules depend on full conformance of these expressions, which depends on the specific declarations involved. 2.a/3

Aspect Description for Pre: Precondition; a condition that is expected to hold true before a call. 2.b/5

{*AI05-0254-1*} {*AI05-0262-1*} {*AI12-0418-1*} Pre'Class 3/5

This aspect specifies a class-wide precondition for a dispatching operation of a tagged type and its descendants; it shall be specified by an expression, called a *class-wide precondition expression*. If not specified for an entity, then if no other class-wide precondition applies to the entity, the class-wide precondition expression for the entity is the enumeration literal True.

3.a/3 **Ramification:** {*AI05-0254-1*} If other class-wide preconditions apply to the entity and no class-wide precondition is specified, no class-wide precondition is defined for the entity; of course, the class-wide preconditions (of ancestors) that apply are still going to be checked. We need subprograms that don't have ancestors and don't specify a class-wide precondition to have a class-wide precondition of True, so that adding such a precondition to a descendant has no effect (necessary as a dispatching call through the root routine would not check any precondition).

3.b/5 {*AI12-0220-1*} Pre'Class cannot be specified on an access-to-subprogram type because of a Legality Rule found in 13.1.1 that limits 'Class aspects to tagged types and primitive subprograms of tagged types. The same is true for Post'Class (below).

3.c/5 **Aspect Description for Pre'Class:** Precondition that applies to corresponding subprograms of descendant types.

4/5 {*AI12-0220-1*} Post

This aspect specifies a specific postcondition for a callable entity or an access-to-subprogram type; it shall be specified by an expression, called a *specific postcondition expression*. If not specified for an entity, the specific postcondition expression for the entity is the enumeration literal True.

4.a/5 **Aspect Description for Post:** Postcondition; a condition that will hold true after a call.

5/5 {*AI05-0262-1*} {*AI12-0418-1*} Post'Class

This aspect specifies a class-wide postcondition for a dispatching operation of a tagged type and its descendants; it shall be specified by an expression, called a *class-wide postcondition expression*. If not specified for an entity, the class-wide postcondition expression for the entity is the enumeration literal True.

5.a/5 **Aspect Description for Post'Class:** Postcondition that applies to corresponding subprograms of descendant types.

5.b/5 **Discussion:** {*AI12-0005-1*} In the AARM notes below, we use the terms “inherited” and “inheritance” informally with respect to class-wide pre/post-conditions, to mean that the aspect applies to corresponding subprograms in descendant types.

5.c/5 **Term entry: precondition** — assertion that is expected to be True when a given subprogram is called

5.d/5 **Term entry: postcondition** — assertion that is expected to be True when a given subprogram returns normally

Name Resolution Rules

6/3 {*AI05-0145-2*} The expected type for a precondition or postcondition expression is any boolean type.

7/5 {*AI05-0145-2*} {*AI05-0262-1*} {*AI12-0113-1*} {*AI12-0159-1*} {*AI12-0170-1*} {*AI12-0418-1*} Within the expression for a Pre'Class or Post'Class aspect for a primitive subprogram *S* of a tagged type *T*, a name that denotes a formal parameter (or *S*'Result) of type *T* is interpreted as though it had a (notional) nonabstract type *NT* that is a formal derived type whose ancestor type is *T*, with directly visible primitive operations. Similarly, a name that denotes a formal access parameter (or *S*'Result for an access result) of type access-to-*T* is interpreted as having type access-to-*NT*. [The result of this interpretation is that the only operations that can be applied to such names are those defined for such a formal derived type.]

7.a/4 **Reason:** {*AI12-0159-1*} This ensures that the expression is well-defined for any primitive subprogram of a type descended from *T*.

7.b/5 **Ramification:** {*AI12-0170-1*} The operations of *NT* are also nonabstract, so the rule against a call of an abstract subprogram does not trigger for a class-wide precondition or postcondition.

8/5 {*AI05-0145-2*} {*AI05-0264-1*} {*AI12-0418-1*} For an attribute_reference with attribute_designator Old, if the attribute reference has an expected type (or class of types) or shall resolve to a given type, the same applies to the prefix; otherwise, the prefix shall be resolved independently of context.

Legality Rules

9/3 {*AI05-0145-2*} {*AI05-0230-1*} The Pre or Post aspect shall not be specified for an abstract subprogram or a null procedure. [Only the Pre'Class and Post'Class aspects may be specified for such a subprogram.]

9.a/3 **Discussion:** {*AI05-0183-1*} Pre'Class and Post'Class can only be specified on primitive routines of tagged types, by a blanket rule found in 13.1.1.

{*AI05-0247-1*} {*AI05-0254-1*} If a type *T* has an implicitly declared subprogram *P* inherited from a parent type *T1* and a homograph (see 8.3) of *P* from a progenitor type *T2*, and 10/3

- the corresponding primitive subprogram *P1* of type *T1* is neither null nor abstract; and 11/3
- the class-wide precondition expression True does not apply to *P1* (implicitly or explicitly); and 12/3
- there is a class-wide precondition expression that applies to the corresponding primitive subprogram *P2* of *T2* that does not fully conform to any class-wide precondition expression that applies to *P1*, 13/3

{*AI05-0247-1*} {*AI05-0254-1*} then: 14/3

- If the type *T* is abstract, the implicitly declared subprogram *P* is *abstract*. 15/3
- Otherwise, the subprogram *P* *requires overriding* and shall be overridden with a nonabstract subprogram. 16/3

Discussion: We use the term "requires overriding" here so that this rule is taken into account when calculating visibility in 8.3; otherwise we would have a mess when this routine is overridden. 16.a/3

Reason: Such an inherited subprogram would necessarily violate the Liskov Substitutability Principle (LSP) if called via a dispatching call from an ancestor other than the one that provides the called body. In such a case, the class-wide precondition of the actual body is stronger than the class-wide precondition of the ancestor. If we did not enforce that precondition for the body, the body could be called when the precondition it knows about is False — such "counterfeiting" of preconditions has to be avoided. But enforcing the precondition violates LSP. We do not want the language to be implicitly creating bodies that violate LSP; the programmer can still write an explicit body that calls the appropriate parent subprogram. In that case, the violation of LSP is explicitly in the code and obvious to code reviewers (both human and automated). 16.b/3

We have to say that the subprogram is abstract for an abstract type in this case, so that the next concrete type has to override it for the reasons above. Otherwise, inserting an extra level of abstract types would eliminate the requirement to override (as there is only one declared operation for the concrete type), and that would be bad for the reasons given above. 16.c/3

Ramification: This requires the set of class-wide preconditions that apply to the interface routine to be strictly stronger than those that apply to the concrete routine. Since full conformance requires each name to denote the same declaration, it is unlikely that independently declared preconditions would conform. This rule does allow "diamond inheritance" of preconditions, and of course no preconditions at all match. 16.d/3

We considered adopting a rule that would allow examples where the expressions would conform after all inheritance has been applied, but this is complex and is not likely to be common in practice. Since the penalty here is just that an explicit overriding is required, the complexity is too much. 16.e/3

{*AI05-0247-1*} If a renaming of a subprogram or entry *S1* overrides an inherited subprogram *S2*, then the overriding is illegal unless each class-wide precondition expression that applies to *S1* fully conforms to some class-wide precondition expression that applies to *S2* and each class-wide precondition expression that applies to *S2* fully conforms to some class-wide precondition expression that applies to *S1*. 17/3

Reason: Such an overriding subprogram would violate LSP, as the precondition of *S1* would usually be different (and thus stronger) than the one known to a dispatching call through an ancestor routine of *S2*. This is always OK if the preconditions match, so we always allow that. 17.a/3

Ramification: This only applies to primitives of tagged types; other routines cannot have class-wide preconditions. 17.b/3

{*AI12-0131-1*} Pre'Class shall not be specified for an overriding primitive subprogram of a tagged type *T* unless the Pre'Class aspect is specified for the corresponding primitive subprogram of some ancestor of *T*. 17.1/4

Reason: Any such Pre'Class will have no effect, as it will be **or**ed with True. As such, it is highly misleading for readers, especially for those who are determining the assumptions that can be made in the body of the primitive subprogram. Note that in this case there is nothing explicit that might indicate that the class-wide precondition is ineffective. This rule does not prevent explicitly writing an ineffective class-wide precondition (for instance, if the parent subprogram has explicitly specified a precondition of True). 17.c/4

{*AI12-0131-1*} In addition to the places where Legality Rules normally apply (see 12.3), these rules also apply in the private part of an instance of a generic unit. 17.2/4

Static Semantics

18/5 {*AI05-0145-2*} {*AI12-0113-1*} {*AI12-0131-1*} {*AI12-0170-1*} If a Pre'Class or Post'Class aspect is specified for a primitive subprogram *S* of a tagged type *T*, or such an aspect defaults to True, then a corresponding expression also applies to the corresponding primitive subprogram *S* of each descendant of *T* [(including *T* itself)]. The *corresponding expression* is constructed from the associated expression as follows:

18.a/4 **Ramification:** A Pre'Class defaults to True only if no class-wide preconditions are inherited for the subprogram. The same is true for Post'Class.

18.b/4 **Reason:** We have to inherit precondition expressions that default to True, so that later overridings don't strengthen the precondition (a violation of LSP). We do the same for postconditions for consistency.

18.1/4 - {*AI12-0113-1*} References to formal parameters of *S* (or to *S* itself) are replaced with references to the corresponding formal parameters of the corresponding inherited or overriding subprogram *S* (or to the corresponding subprogram *S* itself).

18.c/4 **Reason:** We have to define the corresponding expression this way as overriding routines are only required to be subtype conformant; in particular, the parameter names can be different. So we have to talk about corresponding parameters without mentioning any names.

18.2/5 {*AI12-0113-1*} {*AI12-0412-1*} If the primitive subprogram *S* is not abstract, but the given descendant of *T* is abstract, then a nondispatching call on *S* is illegal if any Pre'Class or Post'Class aspect that applies to *S* is other than a static boolean expression. Similarly, a primitive subprogram of an abstract type *T*, to which a non-static Pre'Class or Post'Class aspect applies, shall neither be the prefix of an Access attribute_reference, nor shall it be a generic actual subprogram for a formal subprogram declared by a formal_concrete_subprogram_declaration.

18.d/5 *This paragraph was deleted.*

18.e/5 **Reason:** {*AI12-0412-1*} The above rules mean that a concrete primitive of an abstract type is effectively treated as abstract, if any nontrivial Pre'Class or Post'Class aspects apply to it. This makes sense because we are using a notional formal derived type model for such aspects, and an abstract type is not permitted as an actual type for such a formal type. If we didn't do this, the evaluation of the precondition or postcondition of a concrete subprogram of an abstract type could possibly call abstract functions..

18.f/5 **Discussion:** {*AI12-0412-1*} As this Reference Manual was frozen, a significant incompatibility has come to light with the above rule. The wording makes some calls to non-abstract primitives of a tagged abstract type illegal even if no abstract routines are involved in the Pre'Class or Post'Class. It is likely that the above rule will be adjusted; check with ARG work at www.ada-auth.org/arg.html to find the adjusted rules.

19/5 {*AI05-0145-2*} {*AI05-0262-1*} {*AI05-0290-1*} {*AI12-0220-1*} If performing checks is required by the Pre, Pre'Class, Post, or Post'Class assertion policies (see 11.4.2) in effect at the point of a corresponding aspect specification applicable to a given subprogram, entry, or access-to-subprogram type, then the respective precondition or postcondition expressions are considered *enabled*.

19.a/3 **Ramification:** {*AI05-0290-1*} If a class-wide precondition or postcondition expression is enabled, it remains enabled when inherited by an overriding subprogram, even if the policy in effect is Ignore for the inheriting subprogram.

20/5 {*AI05-0273-1*} {*AI12-0280-2*} A subexpression of a postcondition expression is *known on entry* if it is any of:

21/5 - {*AI12-0280-2*} a static subexpression (see 4.9);

22/5 - {*AI12-0280-2*} a literal whose type does not have any Integer_Literal, Real_Literal, or String_Literal aspect specified, or the function specified by such an attribute has aspect Global specified to be **null**;

22.a/5 **Reason:** {*AI12-0280-2*} We mention literals explicitly in case they are not static (as when their subtype is not static, they are the literal **null**, and so on). We exclude literals of types with the aspects that are not Global => **null** as those cause a user-written subprogram with possible side effects to be called.

22.1/5 - {*AI12-0032-1*} {*AI12-0280-2*} {*AI12-0422-1*} a name statically denoting a full constant declaration which is known to have no variable views (see 3.3);

Ramification: Constants of types with immutably limited or controlled parts are not allowed by this rule. Generic formal in objects are allowed by this rule (as they are defined to be full constant declarations). 22.b/5

Reason: We only want things that cannot be changed. We can't just say "constant" since that includes views of variables in some cases (for instance, a dereference of an access to constant object can be a view of a variable). There are other things we could have allowed (like a loop parameter), but having a subprogram declaration where those could be used (like inside of a loop) seems unusual enough to not be worth defining. 22.c/5

- {*AI12-0280-2*} a name statically denoting a nonaliased **in** parameter of an elementary type; 22.2/5

  Ramification: All such parameters are by-copy, so the value won't change during the execution of the subprogram. 22.d/5

- {*AI12-0280-2*} an Old attribute_reference; 22.3/5
- {*AI12-0280-2*} an invocation of a predefined operator where all of the operands are known on entry; 22.4/5
- {*AI12-0280-2*} a function call where the function has aspect Global => **null** where all of the actual parameters are known on entry; 22.5/5

  Reason: Such a function can only depend on the values of its parameters. 22.e/5

- {*AI12-0280-2*} a selected_component of a known-on-entry prefix; 22.6/5
- {*AI12-0280-2*} an indexed_component of a known-on-entry prefix where all index expressions are known on entry; 22.7/5
- {*AI12-0280-2*} a parenthesized known-on-entry expression; 22.8/5
- {*AI12-0280-2*} a qualified_expression or type_conversion whose operand is a known-on-entry expression; 22.9/5
- {*AI12-0280-2*} a conditional_expression where all of the conditions, *selecting*_expressions, and *dependent*_expressions are known on entry. 22.10/5

  Discussion: It's OK if such an expression raises an exception, so long as every evaluation of the expression raises the same exception. 22.f/5

{*AI12-0198-1*} {*AI12-0280-2*} A subexpression of a postcondition expression is *unconditionally evaluated*, *conditionally evaluated*, or *repeatedly evaluated*. A subexpression is considered unconditionally evaluated unless it is conditionally evaluated or repeatedly evaluated. 22.11/5

{*AI12-0280-2*} The following subexpressions are repeatedly evaluated: 22.12/5

- {*AI12-0280-2*} A subexpression of a predicate of a quantified_expression; 22.13/5
- {*AI12-0280-2*} A subexpression of the expression of an array_component_association; 22.14/5
- {*AI12-0280-2*} A subexpression of the expression of a container_element_association. 22.15/5

{*AI12-0280-2*} For a subexpression that is conditionally evaluated, there is a set of *determining expressions* that determine whether the subexpression is actually evaluated at run time. Subexpressions that are conditionally evaluated and their determining expressions are as follows: 22.16/5

- {*AI12-0280-2*} For an if_expression that is not repeatedly evaluated, a subexpression of any part other than the first condition is conditionally evaluated, and its determining expressions include all conditions of the if_expression that precede the subexpression textually; 22.17/5
- {*AI12-0280-2*} For a case_expression that is not repeatedly evaluated, a subexpression of any *dependent*_expression is conditionally evaluated, and its determining expressions include the *selecting*_expression of the case_expression; 22.18/5
- {*AI12-0280-2*} For a short-circuit control form that is not repeatedly evaluated, a subexpression of the right-hand operand is conditionally evaluated, and its determining expressions include the left-hand operand of the short-circuit control form; 23/5
- {*AI12-0280-2*} For a membership test that is not repeatedly evaluated, a subexpression of a membership_choice other than the first is conditionally evaluated, and its determining 24/5

expressions include the *tested*_simple_expression and the preceding membership_choices of the membership test.

24.1/5 {*AI12-0280-2*} A conditionally evaluated subexpression is *determined to be unevaluated* at run time if its set of determining expressions are all known on entry, and when evaluated on entry their values are such that the given subexpression is not evaluated.

24.a/5 **Ramification:** To be precise, a conditionally evaluated expression is determined to be unevaluated (including all of its subexpressions) under the following circumstances:

24.b/5 - Within an if_expression, a *dependent*_expression with an associated condition that evaluates to False, or a condition or *dependent*_expression where a condition of a preceding part of the if_expression evaluates to True;

24.c - Within a case_expression, a *dependent*_expression with an discrete_choice_list that is not covered by the value of the *selecting*_expression;

24.d/5 - The right-hand operand of a short-circuit control form where the left-hand operand evaluates to False for **and then** or True for **or else**;

24.e/5 - A membership_choice of a membership test where the individual membership test defined by any prior membership_choice evaluates to True.

25/3 {*AI05-0145-2*} For a prefix X that denotes an object of a nonlimited type, the following attribute is defined:

26/5 X'Old {*AI05-0145-2*} {*AI05-0262-1*} {*AI05-0273-1*} {*AI12-0032-1*} {*AI12-0280-2*} Each X'Old in a postcondition expression that is enabled, other than those that occur in subexpressions that are determined to be unevaluated, denotes a constant that is implicitly declared at the beginning of the subprogram body, entry body, or accept statement.

26.a/5 **Ramification:** {*AI12-0280-2*} If X'Old occurs in a subexpression that is determined to be unevaluated, then there is no associated constant, and no evaluation of the prefix takes place. In general, this will require evaluating one or more known-on-entry subexpressions before creating and initializing any X'Old constants. Note that any 'Old in a known-on-entry subexpression evaluated this way represents the current value of the prefix (the 'Old itself can be ignored).

26.b/5 {*AI12-0193-1*} In the case of an accept statement, the constant is declared inside of the rendezvous. It is considered part of the initialization of the postcondition check, which is part of the rendezvous by definition (see 9.5.2).

26.1/4 {*AI12-0032-1*} {*AI12-0159-1*} The implicitly declared entity denoted by each occurrence of X'Old is declared as follows:

26.2/4 - If X is of an anonymous access type defined by an access_definition *A* then

26.3/4
```
X'Old : constant A := X;
```

26.4/4 - If X is of a specific tagged type *T* then

26.5/4
```
anonymous : constant T'Class := T'Class(X);
X'Old : T renames T(anonymous);
```

26.6/4 where the name X'Old denotes the object renaming.

26.c/4 **Ramification:** This means that the underlying tag associated with X'Old is that of X and not that of the nominal type of X.

26.7/4 - Otherwise

26.8/4
```
X'Old : constant S := X;
```

26.9/4 where *S* is the nominal subtype of X. This includes the case where the type of *S* is an anonymous array type or a universal type.

26.10/5 {*AI12-0032-1*} {*AI12-0185-1*} {*AI12-0388-1*} The type and nominal subtype of X'Old are as implied by the above definitions.

27/5 {*AI05-0145-2*} {*AI05-0262-1*} {*AI05-0273-1*} {*AI12-0217-1*} {*AI12-0280-2*} Reference to this attribute is only allowed within a postcondition expression. The prefix of an Old attribute_reference shall not contain a Result attribute_reference, nor an Old attribute_reference, nor a use of an entity declared within the postcondition expression but not within prefix itself (for example, the loop parameter of an enclosing

quantified_expression). The prefix of an Old attribute_reference shall statically name (see 4.9) an entity, unless the attribute_reference is unconditionally evaluated, or is conditionally evaluated where all of the determining expressions are known on entry.

Discussion: The prefix X can be any nonlimited object that obeys the syntax for prefix other than the few exceptions given above (discussed below). Useful cases are: the name of a formal parameter of mode [**in**] **out**, the name of a global variable updated by the subprogram, a function call passing those as parameters, a subcomponent of those things, etc. 27.a/3

A qualified expression can be used to make an arbitrary expression into a valid prefix, so T'(X + Y)'Old is legal, even though (X + Y)'Old is not. The value being saved here is the sum of X and Y (a function result is an object). Of course, in this case "+"(X, Y)'Old is also legal, but the qualified expression is arguably more readable. 27.b/3

Note that F(X)'Old and F(X'Old) are not necessarily equal. The former calls F(X) and saves that value for later use during the postcondition. The latter saves the value of X, and during the postcondition, passes that saved value to F. In most cases, the former is what one wants (but it is not always legal, see below). 27.c/3

{*AI12-0032-1*} If X has controlled parts, adjustment and finalization are implied by the implicit constant declaration. Similarly, the implicit constant declaration defines the accessibility level of X'Old. 27.d/4

If postconditions are disabled, we want the compiler to avoid any overhead associated with saving 'Old values. 27.e/3

'Old makes no sense for limited types, because its implementation involves copying. It might make semantic sense to allow build-in-place, but it's not worth the trouble. 27.f/3

Reason: {*AI05-0273-1*} {*AI12-0280-2*} {*AI12-0005-1*} Since the prefix of an Old attribute_reference is evaluated when the subprogram is called (if it is evaluated at all), we cannot allow it to include values that do not exist at that time (like 'Result and loop parameters of quantified_expressions). We also do not allow the prefix itself to include 'Old references, as those would be redundant (because the evaluation of the prefix, if it occurs, already happens on entry to the subprogram), and allowing them would require some sort of order to the implicit constant declarations (because in A(I'Old)'Old, we surely would want the value of I'Old evaluated before the A(I'Old) is evaluated). 27.g/5

{*AI05-0273-1*} {*AI12-0280-2*} An additional rule applies when it cannot be determined on entry to the subprogram whether the Old attribute_reference will or will not be evaluated when the overall postcondition expression is evaluated. In such cases, we require that the prefix of the Old attribute_reference to statically name some object. This is necessary because the Old prefixes have to be evaluated when the subprogram is called if there is any possibility that they might be needed; the compiler cannot in general know whether they will be needed in the postcondition expression. To see the problem, consider: 27.h/5

```
Table : array (1..10) of Integer := ...                                    27.i/3
procedure Bar (I : in out Natural)
   with Post => I > 0 and then Table(I)'Old = 1; -- Illegal
```

{*AI12-0005-1*} In this example, the compiler cannot know on entry what will be the value of I when the subprogram returns (since the subprogram execution can change it), and thus it does not know whether Table(I)'Old will be needed then. Thus it has to always create an implicit constant and evaluate Table(I) when Bar is called (because not having the value when it is needed is not acceptable). But if I = 0 when the subprogram is called, that evaluation will raise Constraint_Error, and that will happen even if I is unchanged by the subprogram and the value of Table(I)'Old is not ultimately needed. It's easy to see how a similar problem could occur for a dereference of an access type. This would be mystifying (since the point of the short circuit is to eliminate this possibility, but it cannot do so). Therefore, we require the prefix of any Old attribute_reference in such a context to statically name an object, which eliminates anything that could change during execution. 27.j/5

It is easy to work around most errors that occur because of this rule. Just move the 'Old to the outer object, before any indexing, dereferences, or components. (That does not work for function calls, however, nor does it work for array indexing if the index can change during the execution of the subprogram.) 27.k/3

Ramification: {*AI12-0032-1*} An accept statement for a task entry with enabled postconditions such as 27.l/4

```
accept E do                                                                27.m/4
   statements
exception
   handlers
end;
```

behaves (at runtime) as follows: 27.n/4

27.o/4
```
accept E do
   declare
      declarations, if any, of 'Old constants
   begin
      begin
         statements
      exception
         handlers
      end;
      postcondition checks
   end;
end;
```

27.p/4 {*AI12-0032-1*} Preconditions are checked by the caller before the rendezvous begins. Postcondition expressions might, of course, reference 'Old constants.

27.q/4 {*AI12-0032-1*} In the case of a protected operation with enabled postconditions, 'Old constant declarations (if any) are elaborated after the start of the protected action. Postcondition checks (which might reference these constants) are performed before the end of the protected action as described below.

27.r/5 {*AI12-0280-2*} The prefix of an Old attribute_reference has to statically name an entity if it appears within a repeatedly evaluated expression.

28/5 {*AI05-0145-2*} {*AI05-0220-1*} For a prefix F that denotes a function declaration or an access-to-function type, the following attribute is defined:

29/5 F'Result {*AI05-0145-2*} {*AI05-0262-1*} {*AI12-0185-1*} {*AI12-0220-1*} {*AI12-0388-1*} Within a postcondition expression for F, denotes the return object of the function call for which the postcondition expression is evaluated. The type of this attribute is that of the result subtype of the function or access-to-function type except within a Post'Class postcondition expression for a function with a controlling result or with a controlling access result; in those cases the type of the attribute is described above as part of the Name Resolution Rules for Post'Class.

30/3 {*AI05-0262-1*} Use of this attribute is allowed only within a postcondition expression for F.

30.a/5 **To be honest:** {*AI05-0220-1*} An “access-to-function type” is an access-to-subprogram type whose designated profile is a function profile.

30.1/5 {*AI12-0143-1*} For a prefix E that denotes an entry declaration of an entry family (see 9.5.2), the following attribute is defined:

30.2/5 E'Index {*AI12-0143-1*} Within a precondition or postcondition expression for entry family E, denotes the value of the entry index for the call of E. The nominal subtype of this attribute is the entry index subtype.

30.3/5 {*AI12-0143-1*} Use of this attribute is allowed only within a precondition or postcondition expression for E.

Dynamic Semantics

31/3 {*AI05-0145-2*} {*AI05-0247-1*} {*AI05-0290-1*} Upon a call of the subprogram or entry, after evaluating any actual parameters, precondition checks are performed as follows:

32/3 - The specific precondition check begins with the evaluation of the specific precondition expression that applies to the subprogram or entry, if it is enabled; if the expression evaluates to False, Assertions.Assertion_Error is raised; if the expression is not enabled, the check succeeds.

33/3 - The class-wide precondition check begins with the evaluation of any enabled class-wide precondition expressions that apply to the subprogram or entry. If and only if all the class-wide precondition expressions evaluate to False, Assertions.Assertion_Error is raised.

33.a/3 **Ramification:** The class-wide precondition expressions of the entity itself as well as those of any parent or progenitor operations are evaluated, as these expressions apply to the corresponding operations of all descendants.

33.b/3 Class-wide precondition checks are performed for all appropriate calls, but only enabled precondition expressions are evaluated. Thus, the check would be trivial if no precondition expressions are enabled.

{*AI05-0145-2*} {*AI05-0247-1*} {*AI05-0254-1*} {*AI05-0269-1*} {*AI12-0166-1*} The precondition checks are performed in an arbitrary order, and if any of the class-wide precondition expressions evaluate to True, it is not specified whether the other class-wide precondition expressions are evaluated. The precondition checks and any check for elaboration of the subprogram body are performed in an arbitrary order. In a call on a protected operation, the checks are performed before starting the protected action. For an entry call, the checks are performed prior to checking whether the entry is open. 34/5

Reason: We need to explicitly allow short-circuiting of the evaluation of the class-wide precondition check if any expression fails, as it consists of multiple expressions; we don't need a similar permission for the specific precondition check as it consists only of a single expression. Nothing is evaluated for the call after a check fails, as the failed check propagates an exception. 34.a/3

{*AI05-0145-2*} {*AI05-0247-1*} {*AI05-0254-1*} {*AI05-0262-1*} {*AI05-0290-1*} Upon successful return from a call of the subprogram or entry, prior to copying back any by-copy **in out** or **out** parameters, the postcondition check is performed. This consists of the evaluation of any enabled specific and class-wide postcondition expressions that apply to the subprogram or entry. If any of the postcondition expressions evaluate to False, then Assertions.Assertion_Error is raised. The postcondition expressions are evaluated in an arbitrary order, and if any postcondition expression evaluates to False, it is not specified whether any other postcondition expressions are evaluated. The postcondition check, and any constraint or predicate checks associated with **in out** or **out** parameters are performed in an arbitrary order. 35/3

Ramification: The class-wide postcondition expressions of the entity itself as well as those of any parent or progenitor operations are evaluated, as these apply to all descendants; in contrast, only the specific postcondition of the entity applies. Postconditions can always be evaluated inside the invoked body. 35.a/3

{*AI12-0032-1*} For a call to a task entry, the postcondition check is performed before the end of the rendezvous; for a call to a protected operation, the postcondition check is performed before the end of the protected action of the call. The postcondition check for any call is performed before the finalization of any implicitly-declared constants associated (as described above) with Old attribute_references but after the finalization of any other entities whose accessibility level is that of the execution of the callable construct. 35.1/4

Reason: {*AI12-0032-1*} If a postcondition references the implicitly-declared constant associated with an Old attribute, the postcondition must be evaluated before the constant is finalized. One way to think of this is to imagine declaring a controlled object between any implicit "'Old" constant declarations and any explicit declarations, then performing postcondition checks during the finalization of this object. 35.a.1/4

{*AI05-0145-2*} {*AI05-0262-1*} If a precondition or postcondition check fails, the exception is raised at the point of the call[; the exception cannot be handled inside the called subprogram or entry]. Similarly, any exception raised by the evaluation of a precondition or postcondition expression is raised at the point of call. 36/3

{*AI05-0145-2*} {*AI05-0247-1*} {*AI05-0254-1*} {*AI05-0262-1*} {*AI12-0113-1*} {*AI12-0159-1*} For any call to a subprogram or entry *S* (including dispatching calls), the checks that are performed to verify specific precondition expressions and specific and class-wide postcondition expressions are determined by those for the subprogram or entry actually invoked. Note that the class-wide postcondition expressions verified by the postcondition check that is part of a call on a primitive subprogram of type *T* includes all class-wide postcondition expressions originating in any progenitor of *T*[, even if the primitive subprogram called is inherited from a type *T1* and some of the postcondition expressions do not apply to the corresponding primitive subprogram of *T1*]. Any operations within a class-wide postcondition expression that were resolved as primitive operations of the (notional) formal derived type *NT*, are in the evaluation of the postcondition bound to the corresponding operations of the type identified by the controlling tag of the call on *S*.[This applies to both dispatching and non-dispatching calls on *S*.] 37/4

Ramification: This applies to access-to-subprogram calls, dispatching calls, and to statically bound calls. We need this rule to cover statically bound calls as well, as specific pre- and postconditions are not inherited, but the subprogram might be. 37.a/3

37.b/3 For concrete subprograms, we require the original specific postcondition to be evaluated as well as the inherited class-wide postconditions in order that the semantics of an explicitly defined wrapper that does nothing but call the original subprogram is the same as that of an inherited subprogram.

37.c/3 Note that this rule does not apply to class-wide preconditions; they have their own rules mentioned below.

38/4 {*AI05-0145-2*} {*AI05-0247-1*} {*AI05-0254-1*} {*AI12-0113-1*} {*AI12-0159-1*} The class-wide precondition check for a call to a subprogram or entry *S* consists solely of checking the class-wide precondition expressions that apply to the denoted callable entity (not necessarily to the one that is invoked). Any operations within such an expression that were resolved as primitive operations of the (notional) formal derived type *NT* are in the evaluation of the precondition bound to the corresponding operations of the type identified by the controlling tag of the call on *S*.[This applies to both dispatching and non-dispatching calls on *S*.]

38.a/3 **Ramification:** For a dispatching call, we are talking about the Pre'Class(es) that apply to the subprogram that the dispatching call is resolving to, not the Pre'Class(es) for the subprogram that is ultimately dispatched to. The class-wide precondition of the resolved call is necessarily the same or stronger than that of the invoked call. For a statically bound call, these are the same; for an access-to-subprogram, (which has no class-wide preconditions of its own), we check the class-wide preconditions of the invoked routine.

38.a.1/5 {*AI12-0233-1*} Since this check is based on the “callable entity”, it does not depend on the view of the entity. This matters any time the ancestor type (if any) of the partial view differs from the parent type of the full view. In such a case, the view of the callable entity associated with the full view might inherit a Pre'Class while the view of the same callable entity associated with the partial view does not.

38.b/3 **Implementation Note:** These rules imply that logically, class-wide preconditions of routines must be checked at the point of call (other than for access-to-subprogram calls, which must be checked in the body, probably using a wrapper). Specific preconditions that might be called with a dispatching call or via an access-to-subprogram value must be checked inside of the subprogram body. In contrast, the postcondition checks always need to be checked inside the body of the routine. Of course, an implementation can evaluate all of these at the point of call for statically bound calls if the implementation uses wrappers for dispatching bodies and for 'Access values.

38.c/3 There is no requirement for an implementation to generate special code for routines that are imported from outside of the Ada program. That's because there is a requirement on the programmer that the use of interfacing aspects do not violate Ada semantics (see B.1). That includes making pre- and postcondition checks. For instance, if the implementation expects routines to make their own postcondition checks in the body before returning, C code can be assumed to do this (even though that is highly unlikely). That's even though the formal definition of those checks is that they are evaluated at the call site. Note that pre- and postconditions can be very useful for verification tools (even if they aren't checked), because they tell the tool about the expectations on the foreign code that it most likely cannot analyze.

39/5 {*AI12-0195-1*} For the purposes of the above rules, a call on an inherited subprogram is considered to involve a call on a subprogram *S*' whose body consists only of a call (with appropriate conversions) on the non-inherited subprogram *S* from which the inherited subprogram was derived. It is not specified whether class-wide precondition or postcondition expressions that are equivalent (with respect to which non-inherited function bodies are executed) for *S* and *S*' are evaluated once or twice. If evaluated only once, the value returned is used for both associated checks.

39.a/5 **Implementation Note:** If the class-wide pre- and postcondition expressions are equivalent for *S* and *S*' because none of the primitive subprograms called in those expressions were overridden, no wrapper is needed. Otherwise, a wrapper is presumably needed to provide the correct logic.

40/5 {*AI05-0145-2*} {*AI05-0247-1*} {*AI05-0254-1*} {*AI12-0220-1*} For a call via an access-to-subprogram value, precondition and postcondition checks performed are as determined by the subprogram or entry denoted by the prefix of the Access attribute reference that produced the value. In addition, a precondition check of any precondition expression associated with the access-to-subprogram type is performed. Similarly, a postcondition check of any postcondition expression associated with the access-to-subprogram type is performed.

40.a/5 **Implementation Note:** {*AI12-0220-1*} A call via an access-to-subprogram value can be considered equivalent (with respect to dynamic semantics) to a call to a notional "wrapper" subprogram which has the Pre and Post aspects and the profile of the access-to-subprogram type and whose body contains (and returns, in the case of a function) only a call to the designated subprogram. However, other evaluation orders for the checks are allowed beyond those allowed by strictly following this model. This equivalence can be used to determine the appropriate point at which the constant

associated with an Old attribute reference in the Post aspect for an access-to-subprogram type is elaborated and finalized.

Ramification: {*AI12-0220-1*} In the case of type conversion between two access-to-subprogram types, the Pre and Post aspects of the source type of the conversion play no role in any subsequent call via the conversion result; only the Pre and Post aspects of the target type of the conversion are relevant in that case. The same applies in the case of a "conversion" (using the term loosely) which is accomplished by combining a dereference and an Access attribute reference, as in `Some_Pointer.all'Access`. 40.b/5

{*AI12-0272-1*} [For a call on a generic formal subprogram, precondition and postcondition checks performed are as determined by the subprogram or entry denoted by the actual subprogram, along with any specific precondition and specific postcondition of the formal subprogram itself.] 41/5

Proof: {*AI12-0272-1*} This follows from the general Dynamic Semantics rules given above, but we mention it explicitly so that there can be no doubt that it is intended. 41.a/5

To be honest: {*AI12-0371-1*} The specific precondition and postcondition that apply to a generic formal subprogram also apply to any renaming of that subprogram, even if that renaming is visible in the instance and called from outside of the generic instance. 41.b/5

Implementation Permissions

{*AI12-0280-2*} {*AI12-0373-1*} An implementation may evaluate a known-on-entry subexpression of a postcondition expression of an entity at the place where X'Old constants are created for the entity, with the normal evaluation of the postcondition expression, or both. 42/5

Reason: {*AI12-0280-2*} We allow the evaluation of known-on-entry subexpressions when they might be needed to determine whether to create a particular 'Old constant. We allow them to be evaluated later as well, or for the results to be saved somehow. This permission shouldn't matter, as the results ought to be same wherever they are evaluated and there should not be any side effects. The main effect of the permission is to determine when any exceptions caused by such subexpressions may be raised. We never require waiting to determine the value of such subexpressions, even if they aren't used to determine the creation of a constant for 'Old. 42.a/5

NOTE 1 {*AI05-0145-2*} {*AI05-0262-1*} {*AI12-0442-1*} A precondition is checked just before the call. If another task can change any value that the precondition expression depends on, the precondition can evaluate to False within the subprogram or entry body. 43/5

NOTE 2 {*AI12-0312-1*} For an example of the use of these aspects and attributes, see the Streams Subsystem definitions in 13.13.1. 44/5

Extensions to Ada 2005

{*AI05-0145-2*} {*AI05-0230-1*} {*AI05-0247-1*} {*AI05-0254-1*} {*AI05-0262-1*} {*AI05-0273-1*} {*AI05-0274-1*} Pre and Post aspects are new. 44.a/3

Inconsistencies With Ada 2012

{*AI12-0032-1*} **Corrigendum:** The Old attribute is defined more carefully. This changes the nominal subtype and place of declaration of the attribute compared to the published Ada 2012 Reference Manual. In extreme cases, this could change the runtime behavior of the attribute (for instance, the tag might be different). The changes are most likely going to prevent bugs by being more intuitive, but it is possible that a program that previously worked might fail. 44.b/4

{*AI12-0113-1*} {*AI12-0159-1*} **Corrigendum:** Eliminated unintentional redispatching from class-wide preconditions and postconditions. This means that a different body might be evaluated for a statically bound call to a routine that has a class-wide precondition or postcondition. The change means that the behavior of Pre and Pre'Class will be the same for a particular subprogram, and that the known behavior of the operations can be assumed within the body of that subprogram for Pre'Class. We expect that this change will primarily fix bugs, as it will make Pre'Class and Post'Class work more like expected. In the case where redispatching is desired, an explicit conversion to a class-wide type can be used. 44.c/4

{*AI12-0166-1*} **Correction:** Specified that precondition checks always take place before starting a protected action. Original Ada 2012 left this unspecified, so if an implementation made the checks after starting the protected action, and a program depended upon that, the program might fail in a different compiler. But such a program was depending on unspecified behavior anyway, and thus was never portable; as such, such programs should be rare. 44.d/5

{*AI12-0195-1*} **Correction:** Specified that an inherited subprogram check both the original and new versions of a class-wide precondition. If a call on an inherited subprogram fails the original class-wide precondition when it passes the new class-wide precondition, then the call will fail the precondition check whereas it would have passed in original 44.d.1/5

Ada 2012. (A similar possibility exists for class-wide postconditions.) This can only happen if the overriding subprograms somehow fail to follow the guidelines of LSP, so this should be rare (the entire point of class-wide preconditions and postconditions is to use them in cases where LSP is followed).

Incompatibilities With Ada 2012

44.e/5 {*AI12-0045-1*} {*AI12-0005-1*} **Corrigendum:** Precondition and postcondition aspects cannot be specified on instances of generic subprograms (they should be specified on the generic subprogram instead). This was (unintentionally) allowed by the Ada 2012 standard. These are not allowed on instances as there is no corresponding way to add preconditions and postconditions to subprograms declared within the instance of a generic package. Therefore, allowing specification on a subprogram instance could present a maintenance problem in the future if the entity needs to be converted to a generic package (a common conversion).

44.f/4 {*AI12-0131-1*} **Corrigendum:** Pre'Class is no longer allowed to be specified for an overriding primitive subprogram unless there are also inherited class-wide precondittions. This incompatibility prevents cases where the explicit Pre'Class is counterfeited by an implicit class-wide precondition of True. This rule should catch more bugs than it creates; the programmer should have written Pre rather than Pre'Class in this case (or written Pre'Class on the original subprogram, not an overriding). Note that this incompatibility eliminates what otherwise would be an inconsistency with original Ada 2012, where precondition checks that would have previously been made for a statically bound call would no longer be made. That dynamic change was necessary to eliminate cases where the evaluated class-wide precondition on a dispatching call would have been weaker than the class-wide precondition of a statically bound call. (The original Ada 2012 violated the LSP semantics that class-wide preconditions were intended to model.)

44.g/5 {*AI12-0198-1*} **Correction:** A component expression in an array aggregate can now be potentially unevaluated, requiring the prefix to be statically determined. Existing code that uses the Old attribute with a dynamic prefix in such contexts will now be illegal. However, in many cases, the existing code will not do what the programmer is expecting (as Old is evaluated textually, once per occurrence, while array aggregate components are evaluated once per component). In addition, Old is a new Ada 2012 feature, so most Ada legacy code will not contain it. The problem is usually easily fixed by moving Old to an outer object (such as the entire aggregate).

Extensions to Ada 2012

44.h/5 {*AI12-0143-1*} The Index attribute is new.

44.i/5 {*AI12-0217-1*} **Correction:** The prefix of a statically unevaluated Old attribute_reference can be a selected_component or indexed_component. This is considered a correction as the old rule is unintentionally too fierce, rejecting safe cases.

44.j/5 {*AI12-0280-2*} We make no restriction on the prefix of an Old attribute_reference if we can determine when the subprogram is entered (which is the point when Old prefixes are evaluated) whether it will be needed in the evaluation of the postcondition.

44.k/5 {*AI12-0220-1*} {*AI12-0272-1*} Pre and Post can be given on an access-to-subprogram type and on a generic formal subprogram.

44.l/5 {*AI12-0412-1*} **Correction:** We now allow the definition of a concrete subprogram *S* that has applicable Pre'Class or Post'Class expressions that is primitive for an abstract type *T* even when a Pre'Class or Post'Class may call an abstract subprogram. Rather, *S* is treated as if it is abstract (meaning that uses that might require evaluating a statically bound Pre'Class or Post'Class expression are not allowed).

Wording Changes from Ada 2012

44.m/5 {*AI05-0170-1*} **Correction:** Clarified the wording about the meaning of the notional type *NT* and the corresponding expression. Both changes follow from other rules but are nonobvious.

44.n/5 {*AI05-0185-1*} **Correction:** Removed redundant (and sometimes incorrect) wording about the resolution of the Old and Result attributes.

6.1.2 The Global and Global'Class Aspects

1/5 {*AI12-0425-1*} {*AI12-0439-1*} The Global and Global'Class aspects of a program unit are used to identify the objects global to the unit that can be read or written during its execution.

Syntax

2/5 {*AI12-0079-3*} {*AI12-0380-1*} {*AI12-0425-1*} global_aspect_definition ::=
null

| Unspecified
| global_mode global_designator
| (global_aspect_element{; global_aspect_element})

{*AI12-0079-3*} {*AI12-0380-1*} global_aspect_element ::= 3/5
global_mode global_set
| global_mode **all**
| global_mode **synchronized**

global_mode ::= 4/5
basic_global_mode
| extended_global_mode

basic_global_mode ::= **in** | **in out** | **out** 5/5

global_set ::= global_name {, global_name} 6/5

global_designator ::= **all** | **synchronized** | global_name 7/5

{*AI12-0079-1*} {*AI12-0310-1*} global_name ::= *object*_name | *package*_name 8/5

Name Resolution Rules

{*AI12-0079-3*} {*AI12-0425-1*} {*AI12-0427-1*} A global_name shall resolve to statically name an object or a package (including a limited view of a package). 9/5

Static Semantics

{*AI12-0079-3*} For a subprogram, an entry, an access-to-subprogram type, a task unit, a protected unit, or a library package or generic library package, the following language-defined aspect may be specified with an aspect_specification (see 13.1.1): 10/5

{*AI12-0079-3*} {*AI12-0425-1*} Global 11/5
The Global aspect shall be specified with a global_aspect_definition.

Aspect Description for Global: Global object usage contract. 11.a/5

{*AI12-0079-3*} {*AI12-0422-1*} The Global aspect identifies the set of variables (which, for the purposes of this clause, includes all constants except those which are known to have no variable views (see 3.3)) that are global to a callable entity or task body, and that are read or updated as part of the execution of the callable entity or task body. If specified for a protected unit, it refers to all of the protected operations of the protected unit. Constants of any type may also be mentioned in a Global aspect. 12/5

{*AI12-0079-3*} If not specified or otherwise defined below, the aspect defaults to the Global aspect for the enclosing library unit if the entity is declared at library level, and to Unspecified otherwise. If not specified for a library unit, the aspect defaults to `Global => null` for a library unit that is declared Pure, and to `Global => Unspecified` otherwise. 13/5

{*AI12-0079-3*} For a dispatching subprogram, the following language-defined aspect may be specified with an aspect_specification (see 13.1.1): 14/5

{*AI12-0079-3*} {*AI12-0425-1*} Global'Class 15/5
The Global'Class aspect shall be specified with a global_aspect_definition. This aspect identifies an upper bound on the set of variables global to a dispatching operation that can be read or updated as a result of a dispatching call on the operation. If not specified, the aspect defaults to the Global aspect for the dispatching subprogram.

Aspect Description for Global'Class: Global object usage contract inherited on derivation. 15.a/5

16/5 {*AI12-0380-1*} Together, we refer to the Global and Global'Class aspects as *global* aspects.

17/5 {*AI12-0079-3*} A global_aspect_definition defines the Global or Global'Class aspect of some entity. The Global aspect identifies the sets of global variables that can be read, written, or modified as a side effect of executing the operation(s) associated with the entity. The Global'Class aspect associated with a dispatching operation of type *T* represents a restriction on the Global aspect on a corresponding operation of any descendant of type *T*.

18/5 {*AI12-0079-3*} {*AI12-0416-1*} {*AI12-0442-1*} The Global aspect for a callable entity defines the global variables that can be referenced as part of a call on the entity, including any assertion expressions that apply to the call (even if not enabled), such as preconditions, postconditions, predicates, and type invariants.

19/5 {*AI12-0079-3*} {*AI12-0439-1*} The Global aspect for an access-to-subprogram object (or subtype) identifies the global variables that can be referenced when calling via the object (or any object of that subtype) including assertion expressions that apply.

20/5 {*AI12-0079-3*} {*AI12-0405-1*} For a predefined operator of an elementary type, the function representing an enumeration literal, or any other static function (see 4.9), the Global aspect is **null**. For a predefined operator of a composite type, the Global aspect of the operator defaults to that of the enclosing library unit (unless a Global aspect is specified for the type — see H.7).

21/5 {*AI12-0079-3*} {*AI12-0380-1*} The following is defined in terms of operations that are performed by or on behalf of an entity. The rules on operations apply to the entity(s) associated with those operations.

21.a/5 **Discussion:** {*AI12-0380-1*} The operations performed by a callable entity are those associated with the body of the entity. For other kinds of entities (such as subtypes, see H.7), we explicitly list the associated operations.

22/5 {*AI12-0079-3*} The global variables associated with any global_mode can be read as a side effect of an operation. The **in out** and **out** global_modes together identify the set of global variables that can be updated as a side effect of an operation. Creating an access-to-variable value that designates an object is considered an update of the designated object, and creating an access-to-constant value that designates an object is considered a read of the designated object.

23/5 {*AI12-0079-3*} The overall set of objects associated with each global_mode includes all objects identified for the mode in the global_aspect_definition.

24/5 {*AI12-0079-3*} A global_set identifies a *global variable set* as follows:

25/5 - **all** identifies the set of all global variables;

26/5 - **synchronized** identifies the set of all synchronized variables (see 9.10), as well as variables of a composite type all of whose non-discriminant subcomponents are synchronized;

27/5 - global_name{, global_name} identifies the union of the sets of variables identified by the global_names in the list, for the following forms of global_name:

28/5 - *object*_name identifies the specified global variable (or constant);

29/5 - *package*_name identifies the set of all variables declared in the private part or body of the package, or anywhere within a private descendant of the package.

Legality Rules

30/5 {*AI12-0079-1*} Within a global_aspect_definition, a given global_mode shall be specified at most once. Similarly, within a global_aspect_definition, a given entity shall be named at most once by a global_name.

31/5 {*AI12-0079-3*} If an entity (other than a library package or generic library package) has a Global aspect other than Unspecified or **in out all**, then the associated operation(s) shall read only those variables global

to the entity that are within the global variable set associated with the **in**, **in out**, or **out** global_modes, and the operation(s) shall update only those variables global to the entity that are within the global variable set associated with either the **in out** or **out** global_modes. In the absence of the No_Hidden_Indirect_Globals restriction (see H.4), this ignores objects reached via a dereference of an access value. The above rule includes any possible Global effects of calls occurring during the execution of the operation, except for the following excluded calls:

- calls to formal subprograms; 32/5
- calls associated with operations on formal subtypes; 33/5
- calls through formal objects of an access-to-subprogram type; 34/5
- calls through access-to-subprogram parameters; 35/5
- calls on operations with Global aspect Unspecified. 36/5

{*AI12-0079-3*} The possible Global effects of these excluded calls (other than those that are Unspecified) are taken into account by the caller of the original operation, by presuming they occur at least once during its execution. For calls that are not excluded, the possible Global effects of the call are those permitted by the Global aspect of the associated entity, or by its Global'Class aspect if a dispatching call. 37/5

Ramification: For a predefined equality operator of a composite type, the possible Global effects includes those of the equality operations invoked as part of the evaluation of the operator (which might not be predefined and thus might have a different Global specification than the component types). 37.a/5

{*AI12-0079-3*} If a Global aspect other than Unspecified or **in out all** applies to an access-to-subprogram type, then the prefix of an Access attribute_reference producing a value of such a type shall denote a subprogram whose Global aspect is not Unspecified and is *covered* by that of the result type, where a global aspect *G1* is *covered* by a global aspect *G2* if the set of variables that *G1* identifies as readable or updatable is a subset of the corresponding set for *G2*. Similarly on a conversion to such a type, the operand shall be of a named access-to-subprogram type whose Global aspect is covered by that of the target type. 38/5

{*AI12-0079-3*} If an implementation-defined global_mode applies to a given set of variables, an implementation-defined rule determines what sort of references to them are permitted. 39/5

{*AI12-0079-1*} For a subprogram that is a dispatching operation of a tagged type *T*, each mode of its Global aspect shall identify a subset of the variables identified by the corresponding mode, or by the **in out** mode, of the Global'Class aspect of a corresponding dispatching subprogram of any ancestor of *T*, unless the aspect of that ancestor is Unspecified. 40/5

Implementation Permissions

{*AI12-0079-3*} {*AI12-0444-1*} An implementation can allow some references to a constant object which are not accounted for by the Global or Global'Class aspect when it is considered a variable in the above rules, if the implementation can determine that the object is in fact immutable. 41/5

Ramification: In particular, this allows the implementation to violate privacy in order to determine whether a constant needs to be covered by a Global aspect. 41.a/5

{*AI12-0079-3*} Implementations may perform additional checks on calls to operations with an Unspecified Global aspect to ensure that they do not violate any limitations associated with the point of call. 42/5

Discussion: In this sense, `Global => Unspecified` is not permission to violate the caller's Global restrictions. It is rather that the implementor of the subprogram is presuming other means are being used to ensure safety. Note the No_Unspecified_Globals Restriction (H.4), which prevents the use of Unspecified with the Global aspect in a given partition. 42.a/5

43/5 {*AI12-0079-3*} Implementations may extend the syntax or semantics of the Global aspect in an implementation-defined manner[; for example, supporting additional global_modes].

43.a/5 **Implementation defined:** Any extensions of the Global aspect.

43.b/5 **Reason:** This is intended to allow preexisting usages from SPARK 2014 to remain acceptable in conforming implementations, as well as to provide flexibility for future enhancements. Note the word "extend" in this permission; we expect that any aspect usage that conforms with the (other) rules of this subclause will be accepted by any Ada implementation, regardless of any implementation-defined extensions.

44/5 NOTE {*AI12-0312-1*} For an example of the use of these aspects, see the Vector container definition in A.18.2.

Extensions to Ada 2012

44.a/5 {*AI12-0079-1*} {*AI12-0240-6*} {*AI12-0310-1*} The Global and Global'Class aspects are new.

6.2 Formal Parameter Modes

1 [A parameter_specification declares a formal parameter of mode **in**, **in out**, or **out**.]

Static Semantics

2 A parameter is passed either *by copy* or *by reference*. [When a parameter is passed by copy, the formal parameter denotes a separate object from the actual parameter, and any information transfer between the two occurs only before and after executing the subprogram. When a parameter is passed by reference, the formal parameter denotes (a view of) the object denoted by the actual parameter; reads and updates of the formal parameter directly reference the actual parameter object.]

3/3 {*AI05-0142-4*} {*AI05-0262-1*} A type is a *by-copy type* if it is an elementary type, or if it is a descendant of a private type whose full type is a by-copy type. A parameter of a by-copy type is passed by copy, unless the formal parameter is explicitly aliased.

4 A type is a *by-reference type* if it is a descendant of one of the following:

5 • a tagged type;

6 • a task or protected type;

7/3 • {*AI05-0096-1*} an explicitly limited record type;

7.a/3 *This paragraph was deleted.*{*AI05-0096-1*}

8 • a composite type with a subcomponent of a by-reference type;

9 • a private type whose full type is a by-reference type.

10/5 {*AI05-0142-4*} {*AI05-0188-1*} {*AI12-0027-1*} {*AI12-0236-1*} {*AI12-0317-1*} A parameter of a by-reference type is passed by reference, as is an explicitly aliased parameter of any type. Each value of a by-reference type has an associated object. For a value conversion, the associated object is the anonymous result object if such an object is created (see 4.6); otherwise it is the associated object of the operand. In other cases, the object associated with the evaluated operative constituent of the name or expression (see 4.4) determines its associated object.

10.a **Ramification:** By-reference parameter passing makes sense only if there is an object to reference; hence, we define such an object for each case.

10.b Since tagged types are by-reference types, this implies that every value of a tagged type has an associated object. This simplifies things, because we can define the tag to be a property of the object, and not of the value of the object, which makes it clearer that object tags never change.

10.b.1/5 {*AI12-0317-1*} A construct like parenthesized expression or qualified_expression is ignored for the purposes of determining the associated object; for a conditional_expression, it is relevant only in that it determines which *dependent*_expression defines the associated object.

We considered simplifying things even more by making every value (and therefore every expression) have an associated object. After all, there is little semantic difference between a constant object and a value. However, this would cause problems for untagged types. In particular, we would have to do a constraint check on every read of a type conversion (or a renaming thereof) in certain cases. 10.c

{*AI95-00318-02*} We do not want this definition to depend on the view of the type; privateness is essentially ignored for this definition. Otherwise, things would be confusing (does the rule apply at the call site, at the site of the declaration of the subprogram, at the site of the return statement?), and requiring different calls to use different mechanisms would be an implementation burden. 10.d/2

C.6, "Shared Variable Control" says that a composite type with an atomic or volatile subcomponent is a by-reference type, among other things. 10.e

Every value of a limited by-reference type is the value of one and only one limited object. The *associated object* of a value of a limited by-reference type is the object whose value it represents. Two values of a limited by-reference type are the *same* if and only if they represent the value of the same object. 10.f

We say "by-reference" above because these statements are not always true for limited private types whose underlying type is nonlimited (unfortunately). 10.g

{*AI05-0240-1*} For other parameters, it is unspecified whether the parameter is passed by copy or by reference. 11/3

Discussion: {*AI05-0005-1*} There is no need to incorporate the discussion of AI83-00178, which requires pass-by-copy for certain kinds of actual parameters, while allowing pass-by-reference for others. This is because we explicitly indicate that a function creates an anonymous constant object for its result (see 6.5). We also provide a special dispensation for instances of Unchecked_Conversion to return by reference (see 13.9). 11.a/3

Bounded (Run-Time) Errors

{*AI05-0240-1*} If one name denotes a part of a formal parameter, and a second name denotes a part of a distinct formal parameter or an object that is not part of a formal parameter, then the two names are considered *distinct access paths*. If an object is of a type for which the parameter passing mechanism is not specified and is not an explicitly aliased parameter, then it is a bounded error to assign to the object via one access path, and then read the value of the object via a distinct access path, unless the first access path denotes a part of a formal parameter that no longer exists at the point of the second access [(due to leaving the corresponding callable construct).] The possible consequences are that Program_Error is raised, or the newly assigned value is read, or some old value of the object is read. 12/3

Discussion: For example, if we call "P(X => Global_Variable, Y => Global_Variable)", then within P, the names "X", "Y", and "Global_Variable" are all distinct access paths. If Global_Variable's type is neither pass-by-copy nor pass-by-reference, then it is a bounded error to assign to Global_Variable and then read X or Y, since the language does not specify whether the old or the new value would be read. On the other hand, if Global_Variable's type is pass-by-copy, then the old value would always be read, and there is no error. Similarly, if Global_Variable's type is defined by the language to be pass-by-reference, then the new value would always be read, and again there is no error. 12.a

Reason: We are saying *assign* here, not *update*, because updating any subcomponent is considered to update the enclosing object. 12.b

The "still exists" part is so that a read after the subprogram returns is OK. 12.c

If the parameter is of a by-copy type, then there is no issue here — the formal is not a view of the actual. If the parameter is of a by-reference type, then the programmer may depend on updates through one access path being visible through some other access path, just as if the parameter were of an access type. 12.d

Implementation Note: The implementation can keep a copy in a register of a parameter whose parameter-passing mechanism is not specified. If a different access path is used to update the object (creating a bounded error situation), then the implementation can still use the value of the register, even though the in-memory version of the object has been changed. However, to keep the error properly bounded, if the implementation chooses to read the in-memory version, it has to be consistent -- it cannot then assume that something it has proven about the register is true of the memory location. For example, suppose the formal parameter is L, the value of L(6) is now in a register, and L(6) is used in an indexed_component as in "A(L(6)) := 99;", where A has bounds 1..3. If the implementation can prove that the value for L(6) in the register is in the range 1..3, then it need not perform the constraint check if it uses the register value. However, if the memory value of L(6) has been changed to 4, and the implementation uses that memory value, then it had better not alter memory outside of A. 12.e

Note that the rule allows the implementation to pass a parameter by reference and then keep just part of it in a register, or, equivalently, to pass part of the parameter by reference and another part by copy. 12.f

12.g **Reason:** We do not want to go so far as to say that the mere presence of aliasing is wrong. We wish to be able to write the following sorts of things in standard Ada:

12.h
```
procedure Move ( Source  : in  String;
                 Target  : out String;
                 Drop    : in  Truncation := Error;
                 Justify : in  Alignment  := Left;
                 Pad     : in  Character  := Space);
-- Copies elements from Source to Target (safely if they overlap)
```

12.i This is from the standard string handling package. It would be embarrassing if this couldn't be written in Ada!

12.j The "then" before "read" in the rule implies that the implementation can move a read to an earlier place in the code, but not to a later place after a potentially aliased assignment. Thus, if the subprogram reads one of its parameters into a local variable, and then updates another potentially aliased one, the local copy is safe — it is known to have the old value. For example, the above-mentioned Move subprogram can be implemented by copying Source into a local variable before assigning into Target.

12.k For an assignment_statement assigning one array parameter to another, the implementation has to check which direction to copy at run time, in general, in case the actual parameters are overlapping slices. For example:

12.l
```
procedure Copy(X : in out String; Y: String) is
begin
    X := Y;
end Copy;
```

12.m It would be wrong for the compiler to assume that X and Y do not overlap (unless, of course, it can prove otherwise).

13/4 NOTE 1 {*AI12-0056-1*} The mode of a formal parameter describes the direction of information transfer to or from the subprogram_body (see 6.1).

14/4 NOTE 2 A formal parameter of mode **in** is a constant view (see 3.3); it cannot be updated within the subprogram_body.

15/5 NOTE 3 {*AI12-0056-1*} {*AI12-0440-1*} A formal parameter of mode **out** can be uninitialized at the start of the subprogram_body (see 6.4.1).

Extensions to Ada 83

15.a The value of an **out** parameter may be read. An **out** parameter is treated like a declared variable without an explicit initial expression.

Wording Changes from Ada 83

15.b Discussion of copy-in for parts of out parameters is now covered in 6.4.1, "Parameter Associations".

15.c The concept of a by-reference type is new to Ada 95.

15.d We now cover in a general way in 3.7.2 the rule regarding erroneous execution when a discriminant is changed and one of the parameters depends on the discriminant.

Wording Changes from Ada 2005

15.e/3 {*AI05-0096-1*} **Correction:** Corrected so that limited derived types are by-reference only if their parent is.

15.f/3 {*AI05-0142-4*} Defined that explicitly aliased parameters (see 6.1) are always passed by reference.

Wording Changes from Ada 2012

15.g/4 {*AI05-0027-1*} **Corrigendum:** Corrected so that value conversions that are copies are the "associated object" for parameter passing of by-reference types. This can only happen if the conversion is between unrelated non-limited types, and it is necessary just so the correct object is defined.

6.3 Subprogram Bodies

1 [A subprogram_body specifies the execution of a subprogram.]

Syntax

2/3 {*AI95-00218-03*} {*AI05-0183-1*} subprogram_body ::=
[overriding_indicator]
subprogram_specification
[aspect_specification] **is**

declarative_part
begin
handled_sequence_of_statements
end [designator];

If a designator appears at the end of a subprogram_body, it shall repeat the defining_designator of the subprogram_specification. 3

Legality Rules

{*AI12-0444-1*} [In contrast to other bodies,] a subprogram_body is allowed to be defined without it being the completion of a previous declaration[, in which case the body declares the subprogram]. If the body is a completion, it shall be the completion of a subprogram_declaration or generic_subprogram_declaration. The profile of a subprogram_body that completes a declaration shall conform fully to that of the declaration. 4/5

Static Semantics

A subprogram_body is considered a declaration. It can either complete a previous declaration, or itself be the initial declaration of the subprogram. 5

Dynamic Semantics

The elaboration of a nongeneric subprogram_body has no other effect than to establish that the subprogram can from then on be called without failing the Elaboration_Check. 6

Ramification: See 12.2 for elaboration of a generic body. Note that protected subprogram_bodies never get elaborated; the elaboration of the containing protected_body allows them to be called without failing the Elaboration_Check. 6.a

[The execution of a subprogram_body is invoked by a subprogram call.] For this execution the declarative_part is elaborated, and the handled_sequence_of_statements is then executed. 7

Examples

Example of procedure body: 8

```
procedure Push(E : in Element_Type; S : in out Stack) is                    9
begin
   if S.Index = S.Size then
      raise Stack_Overflow;
   else
      S.Index := S.Index + 1;
      S.Space(S.Index) := E;
   end if;
end Push;
```

Example of a function body: 10

```
function Dot_Product(Left, Right : Vector) return Real is                   11
   Sum : Real := 0.0;
begin
   Check(Left'First = Right'First and Left'Last = Right'Last);
   for J in Left'Range loop
      Sum := Sum + Left(J)*Right(J);
   end loop;
   return Sum;
end Dot_Product;
```

Extensions to Ada 83

A renaming_declaration may be used instead of a subprogram_body. 11.a

Wording Changes from Ada 83

11.b The syntax rule for subprogram_body now uses the syntactic category handled_sequence_of_statements.

11.c The declarative_part of a subprogram_body is now required; that doesn't make any real difference, because a declarative_part can be empty.

11.d We have incorporated some rules from RM83-6.5 here.

11.e RM83 forgot to restrict the definition of elaboration of a subprogram_body to nongenerics.

Wording Changes from Ada 95

11.f/2 {*AI95-00218-03*} Overriding_indicator is added to subprogram_body.

Extensions to Ada 2005

11.g/3 {*AI05-0183-1*} An optional aspect_specification can be used in a subprogram_body. This is described in 13.1.1.

6.3.1 Conformance Rules

1 [When subprogram profiles are given in more than one place, they are required to conform in one of four ways: type conformance, mode conformance, subtype conformance, or full conformance.]

Static Semantics

2/1 {*8652/0011*} {*AI95-00117-01*} [As explained in B.1, “Interfacing Aspects”, a *convention* can be specified for an entity.] Unless this document states otherwise, the default convention of an entity is Ada. [For a callable entity or access-to-subprogram type, the convention is called the *calling convention*.] The following conventions are defined by the language:

3/3 - {*AI05-0229-1*} The default calling convention for any subprogram not listed below is *Ada*. [The Convention aspect may be specified to override the default calling convention (see B.1)].

3.a **Ramification:** See also the rule about renamings-as-body in 8.5.4.

4 - The *Intrinsic* calling convention represents subprograms that are “built in” to the compiler. The default calling convention is Intrinsic for the following:

5 - an enumeration literal;

6 - a "/=" operator declared implicitly due to the declaration of "=" (see 6.6);

7 - any other implicitly declared subprogram unless it is a dispatching operation of a tagged type;

8 - an inherited subprogram of a generic formal tagged type with unknown discriminants;

8.a.1/1 **Reason:** Consider:

8.a.2/1
```
package P is
    type Root is tagged null record;
    procedure Proc(X: Root);
end P;
```

8.a.3/1
```
generic
    type Formal(<>) is new Root with private;
package G is
    ...
end G;
```

8.a.4/1
```
package body G is
    ...
    X: Formal := ...;
    ...
    Proc(X); -- This is a dispatching call in Instance, because
             -- the actual type for Formal is class-wide.
    ...
    -- Proc'Access would be illegal here, because it is of
    -- convention Intrinsic, by the above rule.
end G;
```

```
type Actual is new Root with ...;                                    8.a.5/1
procedure Proc(X: Actual);
package Instance is new G(Formal => Actual'Class);
    -- It is legal to pass in a class-wide actual, because Formal
    -- has unknown discriminants.
```

Within Instance, all calls to Proc will be dispatching calls, so Proc doesn't really exist in machine code, so we wish to avoid taking 'Access of it. This rule applies to those cases where the actual type might be class-wide, and makes these Intrinsic, thus forbidding 'Access. 8.a.6/1

- an attribute that is a subprogram; 9
- {*AI95-00252-01*} a subprogram declared immediately within a protected_body; 10/2
- {*AI95-00252-01*} {*AI95-00407-01*} {*AI12-0107-1*} any prefixed view of a subprogram (see 4.1.3) without synchronization kind (see 9.5) By_Entry or By_Protected_Procedure. 10.1/4

  Reason: {*AI12-0005-1*} The profile of a prefixed view is different from the "real" profile of the subprogram (it doesn't have the first parameter), so we don't want to be able to take 'Access of it, as that would require generating a wrapper of some sort. 10.a/5

  {*AI12-0107-1*} We except prefixed views that have synchronization kind By_Protected_Procedure so that they can be used with an access-to-protected-procedure type. These don't require special wrappers (this is the normal form for a protected subprogram call). The By_Entry part is just for consistency (there is no access-to-entry type in Ada). 10.b/4

[The Access attribute is not allowed for Intrinsic subprograms.] 11

Ramification: The Intrinsic calling convention really represents any number of calling conventions at the machine code level; the compiler might have a different instruction sequence for each intrinsic. That's why the Access attribute is disallowed. We do not wish to require the implementation to generate an out of line body for an intrinsic. 11.a

{*AI05-0229-1*} Whenever we wish to disallow the Access attribute in order to ease implementation, we make the subprogram Intrinsic. Several language-defined subprograms have "**with** Convention => Intrinsic;". An implementation might actually implement this as "**with** Import => True, Convention => Intrinsic;", if there is really no body, and the implementation of the subprogram is built into the code generator. 11.b/3

Subprograms declared in protected_bodies will generally have a special calling convention so as to pass along the identification of the current instance of the protected type. The convention is not *protected* since such local subprograms need not contain any "locking" logic since they are not callable via "external" calls; this rule prevents an access value designating such a subprogram from being passed outside the protected unit. 11.c

The "implicitly declared subprogram" above refers to predefined operators (other than the "=" of a tagged type) and the inherited subprograms of untagged types. 11.d

- {*AI12-0107-1*} {*AI12-0159-1*} The default calling convention is *protected* for a protected subprogram, for a prefixed view of a subprogram with a synchronization kind of By_Protected_Procedure, and for an access-to-subprogram type with the reserved word **protected** in its definition. 12/4
- {*AI12-0107-1*} {*AI12-0159-1*} The default calling convention is *entry* for an entry and for a prefixed view of a subprogram with a synchronization kind of By_Entry. 13/4
- {*AI95-00254-01*} {*AI95-00409-01*} {*AI05-0264-1*} {*AI12-0207-1*} The calling convention for an anonymous access-to-subprogram parameter or anonymous access-to-subprogram result is *protected* if the reserved word **protected** appears in its definition; otherwise, it is the convention of the entity that has the parameter or result, unless that entity has convention *protected*, *entry*, or Intrinsic, in which case the convention is Ada. 13.1/5

  Ramification: The calling convention for other anonymous access-to-subprogram types is Ada. 13.a/2
- {*8652/0011*} {*AI95-00117-01*} [If not specified above as Intrinsic, the calling convention for any inherited or overriding dispatching operation of a tagged type is that of the corresponding subprogram of the parent type.] The default calling convention for a new dispatching operation of a tagged type is the convention of the type. 13.2/1

  Reason: The first rule is officially stated in 3.9.2. The second is intended to make interfacing to foreign OOP languages easier, by making the default be that the type and operations all have the same convention. 13.a.1/1

{*AI05-0229-1*} Of these four conventions, only Ada and Intrinsic are allowed as a *convention*_identifier in the specification of a Convention aspect. 14/3

14.a/3 **Discussion:** {*AI05-0229-1*} The names of the *protected* and *entry* calling conventions cannot be used in the specification of Convention. Note that **protected** and **entry** are reserved words.

15/2 {*AI95-00409-01*} Two profiles are *type conformant* if they have the same number of parameters, and both have a result if either does, and corresponding parameter and result types are the same, or, for access parameters or access results, corresponding designated types are the same, or corresponding designated profiles are type conformant.

15.a/2 **Discussion:** {*AI95-00409-01*} For anonymous access-to-object parameters, the designated types have to be the same for type conformance, not the access types, since in general each access parameter has its own anonymous access type, created when the subprogram is called. Of course, corresponding parameters have to be either both access parameters or both not access parameters.

15.b/2 {*AI95-00409-01*} Similarly, for anonymous access-to-subprogram parameters, the designated profiles of the types, not the types themselves, have to be conformant.

16/3 {*AI95-00318-02*} {*AI95-00409-01*} {*AI05-0142-4*} Two profiles are *mode conformant* if:

16.1/3 - {*AI05-0142-4*} {*AI05-0262-1*} they are type conformant; and

16.2/3 - {*AI05-0142-4*} corresponding parameters have identical modes and both or neither are explicitly aliased parameters; and

16.3/3 - {*AI05-0207-1*} for corresponding access parameters and any access result type, the designated subtypes statically match and either both or neither are access-to-constant, or the designated profiles are subtype conformant.

17/3 {*AI05-0239-1*} Two profiles are *subtype conformant* if they are mode conformant, corresponding subtypes of the profile statically match, and the associated calling conventions are the same. The profile of a generic formal subprogram is not subtype conformant with any other profile.

17.a **Ramification:**

18/3 {*AI05-0134-1*} {*AI05-0262-1*} Two profiles are *fully conformant* if they are subtype conformant, if they have access-to-subprogram results whose designated profiles are fully conformant, and for corresponding parameters:

18.1/3 - {*AI05-0262-1*} they have the same names; and

18.2/3 - {*AI05-0046-1*} both or neither have null_exclusions; and

18.3/3 - neither have default_expressions, or they both have default_expressions that are fully conformant with one another; and

18.4/3 - {*AI05-0134-1*} for access-to-subprogram parameters, the designated profiles are fully conformant.

18.a **Ramification:** Full conformance requires subtype conformance, which requires the same calling conventions. However, the calling convention of the declaration and body of a subprogram or entry are always the same by definition.

18.b/3 **Reason:** {*AI05-0046-1*} The part about null_exclusions is necessary to prevent controlling parameters from having different exclusions, as such a parameter is defined to exclude null whether or not an exclusion is given.

18.c/3 {*AI05-0134-1*} The parts about access-to-subprogram parameters and results is necessary to prevent such types from having different default_expressions in the specification and body of a subprogram. If that was allowed, it would be undefined which default_expression was used in a call of an access-to-subprogram parameter.

19 Two expressions are *fully conformant* if, [after replacing each use of an operator with the equivalent function_call:]

20 - each constituent construct of one corresponds to an instance of the same syntactic category in the other, except that an expanded name may correspond to a direct_name (or character_literal) or to a different expanded name in the other; and

20.1/4 - {*AI12-0050-1*} corresponding defining_identifiers occurring within the two expressions are the same; and

- {*AI12-0050-1*} each direct_name, character_literal, and selector_name that is not part of the prefix of an expanded name in one denotes the same declaration as the corresponding direct_name, character_literal, or selector_name in the other, or they denote corresponding declarations occurring within the two expressions; and 21/4

  Ramification: {*AI12-0300-1*} Note that it doesn't say "respectively" because a direct_name can correspond to a selector_name, and vice versa, by the previous bullet. This rule allows the prefix of an expanded name to be removed, or replaced with a different prefix that denotes a renaming of the same entity. However, it does not allow a direct_name or selector_name to be replaced with one denoting a distinct renaming (except for direct_names and selector_names in prefixes of expanded names). Note that calls using operator notation are equivalent to calls using prefix notation. 21.a/5

  Given the following declarations: 21.b

  ```
  package A is                                                                    21.c
      function F(X : Integer := 1) return Boolean;
  end A;
  ```

  ```
  {AI05-0005-1} with A;                                                           21.c.1/3
  package B is
      package A_View renames A;
      function F_View(X : Integer := 9999) return Boolean renames A.F;
  end B;
  ```

  ```
  with A, B; use A, B;                                                            21.d
  procedure Main is ...
  ```

  Within Main, the expressions "F", "A.F", "B.A_View.F", and "A_View.F" are all fully conformant with one another. However, "F" and "F_View" are not fully conformant. If they were, it would be bad news, since the two denoted views have different default_expressions. 21.e

  Discussion: {*AI12-0050-1*} We talk about defining_identifiers and "corresponding declarations" because of the possibility of iterator_specifications occurring within the expressions; each iterator_specification is a separate declaration, which we need to allow, but we do want to require that the defining_identifiers are the same. 21.f/4

- {*8652/0018*} {*AI95-00175-01*} {*AI05-0092-1*} each attribute_designator in one is the same as the corresponding attribute_designator in the other; and 21.1/3

- {*AI12-0342-1*} each primary that is a literal in one is a user-defined literal if and only if the corresponding literal in the other is also a user-defined literal. Furthermore, if neither are user-defined literals then they shall have the same values[, but they may have differing textual representations]; if both are user-defined literals then they shall have the same textual representation. 22/5

  Ramification: {*AI12-0342-1*} This rule applies to character_literals, so even though other rules would allow a rename of a character literal to conform to the literal, this rule prevents that. 22.a/5

  Ramification: Note that the above definition makes full conformance a transitive relation. 22.b

Two known_discriminant_parts are *fully conformant* if they have the same number of discriminants, and discriminants in the same positions have the same names, statically matching subtypes, and default_expressions that are fully conformant with one another. 23

Two discrete_subtype_definitions are *fully conformant* if they are both subtype_indications or are both ranges, the subtype_marks (if any) denote the same subtype, and the corresponding simple_expressions of the ranges (if any) fully conform. 24

Ramification: In the subtype_indication case, any ranges have to *be* corresponding; that is, two subtype_indications cannot conform unless both or neither has a range. 24.a

Discussion: This definition is used in 9.5.2, "Entries and Accept Statements" for the conformance required between the discrete_subtype_definitions of an entry_declaration for a family of entries and the corresponding entry_index_specification of the entry_body. 24.b

{*AI95-00345-01*} {*AI95-00397-01*} The *prefixed view profile* of a subprogram is the profile obtained by omitting the first parameter of that subprogram. There is no prefixed view profile for a parameterless subprogram. For the purposes of defining subtype and mode conformance, the convention of a prefixed view profile is considered to match that of either an entry or a protected operation. 24.1/2

24.c/2 **Discussion:** This definition is used to define how primitive subprograms of interfaces match operations in task and protected type definitions (see 9.1 and 9.4).

24.d/2 **Reason:** The weird rule about conventions is pretty much required for synchronized interfaces to make any sense. There will be wrappers all over the place for interfaces anyway. Of course, this doesn't imply that entries have the same convention as protected operations.

Implementation Permissions

25 An implementation may declare an operator declared in a language-defined library unit to be intrinsic.

26/5 NOTE {*AI12-0398-1*} Any conformance requirements between aspect_specifications that are part of a profile or known_discriminant_part are defined by the semantics of each particular aspect. In particular, there is no general requirement for aspect_specifications to match in conforming profiles or discriminant parts.

Extensions to Ada 83

26.a The rules for full conformance are relaxed — they are now based on the structure of constructs, rather than the sequence of lexical elements. This implies, for example, that "(X, Y: T)" conforms fully with "(X: T; Y: T)", and "(X: T)" conforms fully with "(X: **in** T)".

Wording Changes from Ada 95

26.b/2 {*8652/0011*} {*AI95-00117-01*} **Corrigendum:** Clarified that the default convention is Ada. Also clarified that the convention of a primitive operation of a tagged type is the same as that of the type.

26.c/2 {*8652/0018*} {*AI95-00175-01*} **Corrigendum:** Added wording to ensure that two attributes conform only if they have the same attribute_designator.

26.d/2 {*AI95-00252-01*} {*AI95-00254-01*} {*AI95-00407-01*} Defined the calling convention for anonymous access-to-subprogram types and for prefixed views of subprograms (see 4.1.3).

26.e/2 {*AI95-00318-02*} Defined the conformance of access result types (see 6.1).

26.f/2 {*AI95-00345-01*} {*AI95-00397-01*} Defined the prefixed view profile of subprograms for later use.

26.g/2 {*AI95-00409-01*} Defined the conformance of anonymous access-to-subprogram parameters.

Incompatibilities With Ada 2005

26.h/3 {*AI05-0046-1*} **Correction:** Now require null_exclusions to match for full conformance. While this is technically incompatible with Ada 2005 as defined by Amendment 1, it is a new Ada 2005 feature and it is unlikely that users have been intentionally taking advantage of the ability to write mismatching exclusions. In any case, it is easy to fix: add a null_exclusion where needed for conformance.

26.i/3 {*AI05-0134-1*} **Correction:** Now require full conformance of anonymous access-to-subprogram parameters and results for full conformance. This is necessary so that there is no confusion about the default expression that is used for a call. While this is technically incompatible with Ada 2005 as defined by Amendment 1, it is a new Ada 2005 feature and it is unlikely that users have been intentionally taking advantage and writing different default expressions. In any case, it is easy to fix: change any default expressions that don't conform so that they do conform.

26.j/3 {*AI05-0207-1*} **Correction:** Now include the presence or absence of **constant** in access parameters to be considered when checking mode conformance. This is necessary to prevent modification of constants. While this is technically incompatible with Ada 2005 as defined by Amendment 1, it is a new Ada 2005 feature and it is unlikely that users have been intentionally taking advantage and writing mismatching access types.

Wording Changes from Ada 2005

26.k/3 {*AI05-0142-4*} Explicitly aliased parameters are included as part of mode conformance (since it affects the parameter passing mechanism).

Incompatibilities With Ada 2012

26.l/5 {*AI12-0207-1*} **Correction:** The convention of an anonymous access-to-subprogram parameter of a protected entry or subprogram is Ada; if one wants it to be *protected* it can be declared with the keyword **protected**. This is incompatible, but only in a very rare case; usually the intent is to pass a normal subprogram to a protected subprogram (and this was impossible in Ada 2012).

Extensions to Ada 2012

26.m/4 {*AI12-0107-1*} {*AI12-0159-1*} **Corrigendum:** We now define that a prefixed view of a subprogram with synchronization kind By_Protected_Procedure can be used as the prefix of 'Access for an access-to-protected type. We

consider this a correction as it certainly appears that it ought to work, but in original Ada 2012 it would have had a convention mismatch.

Wording Changes from Ada 2012

{*AI12-0050-1*} **Corrigendum:** We now define how two expressions containing quantified expressions can fully conform. This isn't incompatible, as the original Ada 2012 never allowed such expressions to conform (the declarations in each formally being different). Neither is it an extension as one would expect these to conform. 26.n/4

{*AI12-0342-1*} The conformance of literals is tightened up to not assume anything about the value of a user-defined literal. 26.o/5

6.3.2 Inline Expansion of Subprograms

[Subprograms may be expanded in line at the call site.] 1

Paragraphs 2 through 4 were moved to Annex J, "Obsolescent Features".

Static Semantics

{*AI05-0229-1*} For a callable entity or a generic subprogram, the following language-defined representation aspect may be specified: 5/3

Inline — The type of aspect Inline is Boolean. When aspect Inline is True for a callable entity, inline expansion is desired for all calls to that entity. When aspect Inline is True for a generic subprogram, inline expansion is desired for all calls to all instances of that generic subprogram. 5.1/3

If directly specified, the aspect_definition shall be a static expression. [This aspect is never inherited;] if not directly specified, the aspect is False. 5.2/3

Aspect Description for Inline: For efficiency, Inline calls are requested for a subprogram. 5.a/3

This paragraph was deleted.{*AI05-0229-1*} 5.b/3

`This paragraph was deleted.` 5.c/3

`This paragraph was deleted.` 5.d/3

Ramification: {*AI05-0229-1*} The meaning of a subprogram can be changed by inline expansion as requested by aspect Inline only in the presence of failing checks (see 11.6). 5.e/3

Implementation Permissions

{*AI05-0229-1*} For each call, an implementation is free to follow or to ignore the recommendation determined by the Inline aspect. 6/3

Ramification: Note, in particular, that the recommendation cannot always be followed for a recursive call, and is often infeasible for entries. Note also that the implementation can inline calls even when no such desire was expressed via the Inline aspect, so long as the semantics of the program remains unchanged. 6.a

Incompatibilities With Ada 83

This paragraph was deleted.{*AI95-00309-01*} {*AI05-0229-1*} 7.a/3

Extensions to Ada 83

This paragraph was deleted.{*AI05-0229-1*} 7.b/3

Extensions to Ada 95

This paragraph was deleted.{*AI95-00309-01*} {*AI05-0229-1*} 7.c/3

Extensions to Ada 2005

{*AI05-0229-1*} Aspect Inline is new; pragma Inline is now obsolescent. 7.d/3

6.4 Subprogram Calls

1 A *subprogram call* is either a procedure_call_statement or a function_call; [it invokes the execution of the subprogram_body. The call specifies the association of the actual parameters, if any, with formal parameters of the subprogram.]

Syntax

2 procedure_call_statement ::=
*procedure_*name;
| *procedure_*prefix actual_parameter_part;

3 function_call ::=
*function_*name
| *function_*prefix actual_parameter_part

3.a/3 **To be honest:** {*AI05-0005-1*} For the purpose of non-syntax rules, infix operator calls are considered function_calls. See 6.6.

4 actual_parameter_part ::=
(parameter_association {, parameter_association})

5 parameter_association ::=
[*formal_parameter_*selector_name =>] explicit_actual_parameter

6 explicit_actual_parameter ::= expression | *variable_*name

7/5 {*AI12-0416-1*} A parameter_association is *named* or *positional* according to whether or not the *formal_parameter_*selector_name is specified. For the parameter_associations of a single actual_parameter_part or iterator_actual_parameter_part, any positional associations shall precede any named associations. Named associations are not allowed if the prefix in a subprogram call is an attribute_reference.

7.a **Ramification:** This means that the formal parameter names used in describing predefined attributes are to aid presentation of their semantics, but are not intended for use in actual calls.

Name Resolution Rules

8/2 {*AI95-00310-01*} The name or prefix given in a procedure_call_statement shall resolve to denote a callable entity that is a procedure, or an entry renamed as (viewed as) a procedure. The name or prefix given in a function_call shall resolve to denote a callable entity that is a function. The name or prefix shall not resolve to denote an abstract subprogram unless it is also a dispatching subprogram. [When there is an actual_parameter_part, the prefix can be an implicit_dereference of an access-to-subprogram value.]

8.a.1/2 **Discussion:** {*AI95-00310-01*} This rule is talking about dispatching operations (which is a static concept) and not about dispatching calls (which is a dynamic concept).

8.a **Ramification:** The function can be an operator, enumeration literal, attribute that is a function, etc.

9 A subprogram call shall contain at most one association for each formal parameter. Each formal parameter without an association shall have a default_expression (in the profile of the view denoted by the name or prefix). [This rule is an overloading rule (see 8.6).]

9.a/3 **Proof:** {*AI05-0240-1*} All Name Resolution Rules are overloading rules, see 8.6.

Static Semantics

9.1/5 {*AI95-00407-01*} {*AI12-0416-1*} If the name or prefix of a subprogram call denotes a prefixed view (see 4.1.3), the subprogram call is equivalent to a call on the underlying subprogram, with the first actual

parameter being provided by the prefix of the prefixed view (or the Access attribute of this prefix if the first formal parameter is an access parameter), and the remaining actual parameters given by the actual_parameter_part, if any.

Dynamic Semantics

{*AI95-00345-01*} For the execution of a subprogram call, the name or prefix of the call is evaluated, and each parameter_association is evaluated (see 6.4.1). If a default_expression is used, an implicit parameter_association is assumed for this rule. These evaluations are done in an arbitrary order. The subprogram_body is then executed, or a call on an entry or protected subprogram is performed (see 3.9.2). Finally, if the subprogram completes normally, then after it is left, any necessary assigning back of formal to actual parameters occurs (see 6.4.1). 10/2

Discussion: The implicit association for a default is only for this run-time rule. At compile time, the visibility rules are applied to the default at the place where it occurs, not at the place of a call. 10.a

To be honest: If the subprogram is inherited, see 3.4, “Derived Types and Classes”. 10.b

If the subprogram is protected, see 9.5.1, “Protected Subprograms and Protected Actions”. 10.c

If the subprogram is really a renaming of an entry, see 9.5.3, “Entry Calls”. 10.d

{*AI95-00345-01*} If the subprogram is implemented by an entry or protected subprogram, it will be treated as a dispatching call to the corresponding entry (see 9.5.3, “Entry Calls”) or protected subprogram (see 9.5.1, “Protected Subprograms and Protected Actions”). 10.d.1/2

{*AI95-00348-01*} Normally, the subprogram_body that is executed by the above rule is the one for the subprogram being called. For an enumeration literal, implicitly declared (but noninherited) subprogram, null procedure, or an attribute that is a subprogram, an implicit body is assumed. For a dispatching call, 3.9.2, “Dispatching Operations of Tagged Types” defines which subprogram_body is executed. 10.e/2

{*AI95-00318-02*} The exception Program_Error is raised at the point of a function_call if the function completes normally without executing a return statement. 11/2

Discussion: We are committing to raising the exception at the point of call, for uniformity — see AI83-00152. This happens after the function is left, of course. 11.a

Note that there is no name for suppressing this check, since the check imposes no time overhead and minimal space overhead (since it can usually be statically eliminated as dead code). 11.b

{*AI95-00231-01*} A function_call denotes a constant, as defined in 6.5; the nominal subtype of the constant is given by the nominal subtype of the function result. 12/2

Examples

Examples of procedure calls: 13

```
Traverse_Tree;                                          -- see 6.1
Print_Header(128, Title, True);                         -- see 6.1
```
14

```
Switch(From => X, To => Next);                          -- see 6.1
Print_Header(128, Header => Title, Center => True);     -- see 6.1
Print_Header(Header => Title, Center => True, Pages => 128); -- see 6.1
```
15

Examples of function calls: 16

```
Dot_Product(U, V)    -- see 6.1 and 6.3
Clock                -- see 9.6
F.all                -- presuming F is of an access-to-subprogram type — see 3.10
```
17

Examples of procedures with default expressions: 18

```
procedure Activate(Process : in Process_Name;
                   After   : in Process_Name := No_Process;
                   Wait    : in Duration := 0.0;
                   Prior   : in Boolean := False);
```
19

{*AI05-0299-1*} `procedure Pair(Left, Right : in Person_Name := new Person(M));` -- *see 3.10.1* 20/3

21 *Examples of their calls:*

22
```
Activate(X);
Activate(X, After => Y);
Activate(X, Wait => 60.0, Prior => True);
Activate(X, Y, 10.0, False);
```

23/3
```
{AI05-0299-1} Pair;
Pair(Left => new Person(F), Right => new Person(M));
```

24 NOTE If a default_expression is used for two or more parameters in a multiple parameter_specification, the default_expression is evaluated once for each omitted parameter. Hence in the above examples, the two calls of Pair are equivalent.

Examples

25 *Examples of overloaded subprograms:*

26
```
procedure Put(X : in Integer);
procedure Put(X : in String);
```

27
```
procedure Set(Tint   : in Color);
procedure Set(Signal : in Light);
```

28 *Examples of their calls:*

29
```
Put(28);
Put("no possible ambiguity here");
```

30
```
Set(Tint   => Red);
Set(Signal => Red);
Set(Color'(Red));
```

31/5
```
{AI12-0440-1} -- Set(Red) would be ambiguous since Red can
-- denote a value either of type Color or of type Light
```

Wording Changes from Ada 83

31.a We have gotten rid of parameters “of the form of a type conversion” (see RM83-6.4.1(3)). The new view semantics of type_conversions allows us to use normal type_conversions instead.

31.b We have moved wording about run-time semantics of parameter associations to 6.4.1.

31.c We have moved wording about raising Program_Error for a function that falls off the end to here from RM83-6.5.

Extensions to Ada 95

31.d/2 {*AI95-00310-01*} Nondispatching abstract operations are no longer considered when resolving a subprogram call. That makes it possible to use **abstract** to “undefine” a predefined operation for an untagged type. That's especially helpful when defining custom arithmetic packages.

Wording Changes from Ada 95

31.e/2 {*AI95-00231-01*} Changed the definition of the nominal subtype of a function_call to use the nominal subtype wording of 6.1, to take into account null_exclusions and access result types.

31.f/2 {*AI95-00345-01*} Added wording to clarify that the meaning of a call on a subprogram “implemented by” an entry or protected operation is defined by 3.9.2.

31.g/2 {*AI95-00407-01*} Defined the meaning of a call on a prefixed view of a subprogram (see 4.1.3).

Wording Changes from Ada 2012

31.h/5 {*AI12-0416-1*} Moved the equivalence definition for a prefixed view to Static Semantics, in order that it is clear Legality Rules that apply to calls and the parameters of calls apply to prefixed views (including the prefix).

6.4.1 Parameter Associations

1 [A parameter association defines the association between an actual parameter and a formal parameter.]

Language Design Principles

The parameter passing rules for **out** parameters are designed to ensure that the parts of a type that have implicit initial values (see 3.3.1) don't become "de-initialized" by being passed as an **out** parameter. 1.a

{*AI05-0142-4*} For explicitly aliased parameters of functions, we will ensure at the call site that a part of the parameter can be returned as part of the function result without creating a dangling pointer. We do this with accessibility checks at the call site that all actual objects of explicitly aliased parameters live at least as long as the function result; then we can allow them to be returned as access discriminants or anonymous access results, as those have the master of the function result. 1.b/3

Name Resolution Rules

{*AI05-0118-1*} The *formal_parameter*_selector_name of a named parameter_association shall resolve to denote a parameter_specification of the view being called; this is the formal parameter of the association. The formal parameter for a positional parameter_association is the parameter with the corresponding position in the formal part of the view being called. 2/3

To be honest: {*AI05-0118-1*} For positional parameters, the "corresponding position" is calculated after any transformation of prefixed views. 2.a/3

The *actual parameter* is either the explicit_actual_parameter given in a parameter_association for a given formal parameter, or the corresponding default_expression if no parameter_association is given for the formal parameter. The expected type for an actual parameter is the type of the corresponding formal parameter. 3

To be honest: The corresponding default_expression is the one of the corresponding formal parameter in the profile of the view denoted by the name or prefix of the call. 3.a

If the mode is **in**, the actual is interpreted as an expression; otherwise, the actual is interpreted only as a name, if possible. 4

Ramification: {*AI12-0005-1*} This formally resolves the ambiguity present in the syntax rule for explicit_actual_parameter. This matters as an expression that is a name is evaluated and represents a value while a name by itself can be an object; if the mode is not **in**, we want the parameter to interpreted as an object. Note that we don't actually require that the actual be a name if the mode is not **in**; we do that below. 4.a/4

{*AI12-0005-1*} This wording uses "interpreted as" rather than "shall be" so that this rule is not used to resolve overloading; it is solely about evaluation as described above. We definitely do not want to allow oddities like the presence of parentheses requiring the selection of an **in** formal parameter as opposed to an otherwise matching **in out** parameter. 4.b/4

Legality Rules

If the mode is **in out** or **out**, the actual shall be a name that denotes a variable. 5

Discussion: {*AI12-0005-1*} We no longer need "or a type_conversion whose argument is the name of a variable", because a type_conversion is now a name, and a view conversion of a variable is a variable while any other conversion (which should not be legal here) is a constant. 5.a/5

Reason: The requirement that the actual be a (variable) name is not an overload resolution rule, since we don't want the difference between expression and name to be used to resolve overloading. For example: 5.b

```
procedure Print(X : in Integer; Y : in Boolean := True);
procedure Print(Z : in out Integer);
. . .
Print(3); -- Ambiguous!
```
5.c

The above call to Print is ambiguous even though the call is not compatible with the second Print which requires an actual that is a (variable) name ("3" is an expression, not a name). This requirement is a legality rule, so overload resolution fails before it is considered, meaning that the call is ambiguous. 5.d

{*AI12-0074-1*} {*AI12-0159-1*} {*AI12-0377-1*} If the mode is **out**, the actual parameter is a view conversion, and the type of the formal parameter is a scalar type, then 5.1/5

- neither the target type nor the operand type has the Default_Value aspect specified; or 5.2/5

5.3/5 • both the target type and the operand type shall have the Default_Value aspect specified, and there shall exist a type (other than a root numeric type) that is an ancestor of both the target type and the operand type.

5.4/4 {*AI12-0074-1*} {*AI12-0159-1*} In addition to the places where Legality Rules normally apply (see 12.3), these rules also apply in the private part of an instance of a generic unit.

5.e/4 **Reason:** These rules are needed in order to ensure that a well-defined parameter value is passed.

6/3 {*AI05-0102-1*} {*AI05-0142-4*} If the formal parameter is an explicitly aliased parameter, the type of the actual parameter shall be tagged or the actual parameter shall be an aliased view of an object. Further, if the formal parameter subtype *F* is untagged:

6.1/3 • the subtype *F* shall statically match the nominal subtype of the actual object; or

6.2/3 • the subtype *F* shall be unconstrained, discriminated in its full view, and unconstrained in any partial view.

6.a/3 **Ramification:** Tagged objects (and tagged aggregates for **in** parameters) do not need to be aliased. This matches the behavior of unaliased formal parameters of tagged types, which allow 'Access to be taken of the formal parameter regardless of the form of the actual parameter.

6.b/3 **Reason:** We need the subtype check on untagged actual parameters so that the requirements of 'Access are not lost. 'Access makes its checks against the nominal subtype of its prefix, and parameter passing can change that subtype. But we don't want this parameter passing to change the objects that would be allowed as the prefix of 'Access. This is particularly important for arrays, where we don't want to require any additional implementation burden.

6.b.1/5 **Discussion:** {*AI12-0095-1*} {*AI12-0005-1*} We assume the worst in a generic body regarding whether a formal subtype has a constrained partial view; specifically, in a generic body a discriminated subtype is considered to have a constrained partial view if it is a descendant of an untagged generic formal private or derived type (see 12.5.1 for the formal definition of this rule).

6.3/4 {*AI12-0095-1*} In addition to the places where Legality Rules normally apply (see 12.3), these rules also apply in the private part of an instance of a generic unit.

6.4/3 {*AI05-0142-4*} {*AI05-0234-1*} In a function call, the accessibility level of the actual object for each explicitly aliased parameter shall not be statically deeper than the accessibility level of the master of the call (see 3.10.2).

6.c/3 **Discussion:** Since explicitly aliased parameters are either tagged or required to be objects, there is always an object (possibly anonymous) to talk about. This is discussing the static accessibility level of the actual object; it does not depend on any runtime information (for instance when the actual object is a formal parameter of another subprogram, it does not depend on the actual parameter of that other subprogram).

6.d/4 **Ramification:** {*AI12-0095-1*} This accessibility check (and its dynamic cousin as well) can only fail if the master of the function call (which is defined in the Heart of Darkness, or 3.10.2 if you prefer) is different than the master directly enclosing the call. The most likely place where this will occur is in the initializer of an allocator; in almost all other cases this check will always pass.

6.5/3 {*AI05-0144-2*} Two names are *known to denote the same object* if:

6.6/3 • both names statically denote the same stand-alone object or parameter; or

6.7/3 • both names are selected_components, their prefixes are known to denote the same object, and their selector_names denote the same component; or

6.8/3 • both names are dereferences (implicit or explicit) and the dereferenced names are known to denote the same object; or

6.9/3 • both names are indexed_components, their prefixes are known to denote the same object, and each of the pairs of corresponding index values are either both static expressions with the same static value or both names that are known to denote the same object; or

6.10/3 • both names are slices, their prefixes are known to denote the same object, and the two slices have statically matching index constraints; or

- one of the two names statically denotes a renaming declaration whose renamed *object*_name is known to denote the same object as the other, the prefix of any dereference within the renamed *object*_name is not a variable, and any expression within the renamed *object*_name contains no references to variables nor calls on nonstatic functions. 6.11/3

Reason: This exposes known renamings of slices, indexing, and so on to this definition. In particular, if we have 6.e/3

```
C : Character renames S(1);
```
6.f/3

then C and S(1) are known to denote the same object. 6.g/3

We need the requirement that no variables occur in the prefixes of dereferences and in (index) expressions of the renamed object in order to avoid problems from later changes to those parts of renamed names. Consider: 6.h/3

```
   type Ref is access Some_Type;
   Ptr : Ref := new Some_Type'(...);
   X : Some_Type renames Ptr.all;
begin
   Ptr := new Some_Type'(...);
   P (Func_With_Out_Params (Ptr.all), X);
```
6.i/3

X and Ptr.**all** should not be known to denote the same object, since they denote different allocated objects (and this is not an unreasonable thing to do). 6.j/3

To be honest: The exclusion of variables from renamed object_names is not enough to prevent altering the value of the name or expression by another access path. For instance, both **in** parameters passed by reference and access-to-constant values can designate variables. For the intended use of "known to be the same object", this is OK; the modification via another access path is very tricky and it is OK to reject code that would be buggy except for the tricky code. Assuming Element is an elementary type, consider the following example: 6.k/3

```
Global : Tagged_Type;
```
6.l/3

```
procedure Foo (Param : in Tagged_Type := Global) is
   X : Element renames Some_Global_Array (Param.C);
begin
   Global.C := Global.C + 1;
   Swap (X, Some_Global_Array (Param.C));
```
6.m/3

The rules will flag the call of procedure Swap as illegal, since X and Some_Global_Array (Parameter.C) are known to denote the same object (even though they will actually represent different objects if Param = Global). But this is only incorrect if the parameter actually is Global and not some other value; the error could exist for some calls. So this flagging seems harmless. 6.n/3

Similar examples can be constructed using stand-alone composite constants with controlled or immutably limited components, and (as previously noted) with dereferences of access-to-constant values. Even when these examples flag a call incorrectly, that call depends on very tricky code (modifying the value of a constant); the code is likely to confuse future maintainers as well and thus we do not mind rejecting it. 6.o/3

Discussion: Whether or not names or prefixes are known to denote the same object is determined statically. If the name contains some dynamic portion other than a dereference, indexed_component, or slice, it is not "known to denote the same object". 6.p/3

These rules make no attempt to handle slices of objects that are known to be the same when the slices have dynamic bounds (other than the trivial case of bounds being defined by the same subtype), even when the bounds could be proven to be the same, as it is just too complex to get right and these rules are intended to be conservative. 6.q/3

Ramification: "Known to denote the same object" is intended to be an equivalence relationship, that is, it is reflexive, symmetric, and transitive. We believe this follows from the rules. For instance, given the following declarations: 6.r/3

```
S   : String(1..10);
ONE : constant Natural := 1;
R   : Character renames S(1);
```
6.s/3

the names R and S(1) are known to denote the same object by the sixth bullet, and S(1) and S(ONE) are known to denote the same object by the fourth bullet, so using the sixth bullet on R and S(ONE), we simply have to test S(1) vs. S(ONE), which we already know denote the same object. 6.t/3

{*AI05-0144-2*} Two names are *known to refer to the same object* if 6.12/3

- The two names are known to denote the same object; or 6.13/3
- One of the names is a selected_component, indexed_component, or slice and its prefix is known to refer to the same object as the other name; or 6.14/3
- One of the two names statically denotes a renaming declaration whose renamed *object*_name is known to refer to the same object as the other name. 6.15/3

6.u/3 **Reason:** This ensures that names Prefix.Comp and Prefix are known to refer to the same object for the purposes of the rules below. This intentionally does not include dereferences; we only want to worry about accesses to the same object, and a dereference changes the object in question. (There is nothing shared between an access value and the object it designates.)

6.16/3 {*AI05-0144-2*} If a call *C* has two or more parameters of mode **in out** or **out** that are of an elementary type, then the call is legal only if:

6.17/5 - {*AI12-0216-1*} {*AI12-0324-1*} For each name *N* denoting an object of an elementary type that is passed as a parameter of mode **in out** or **out** to the call *C*, there is no other name among the other parameters of mode **in out** or **out** to *C* that is known to denote the same object.

6.v/3 **To be honest:** This means *visibly* an elementary type; it does not include partial views of elementary types (partial views are always composite). That's necessary to avoid having Legality Rules depend on the contents of the private part.

6.18/3 {*AI05-0144-2*} If a construct *C* has two or more direct constituents that are names or expressions whose evaluation may occur in an arbitrary order, at least one of which contains a function call with an **in out** or **out** parameter, then the construct is legal only if:

6.w/3 **Ramification:** All of the places where the language allows an arbitrary order can be found by looking in the index under "arbitrary order, allowed". Note that this listing includes places that don't involve names or expressions (such as checks or finalization).

6.19/3 - For each name *N* that is passed as a parameter of mode **in out** or **out** to some inner function call *C2* (not including the construct *C* itself), there is no other name anywhere within a direct constituent of the construct *C* other than the one containing *C2*, that is known to refer to the same object.

6.x/3 **Ramification:** This requirement cannot fail for a procedure or entry call alone; there must be at least one function with an **in out** or **out** parameter called as part of a parameter expression of the call in order for it to fail.

6.y/3 **Reason:** These rules prevent obvious cases of dependence on the order of evaluation of names or expressions. Such dependence is usually a bug, and in any case, is not portable to another implementation (or even another optimization setting).

6.z/3 In the case that the top-level construct C is a call, these rules do not require checks for most **in out** parameters, as the rules about evaluation of calls prevent problems. Similarly, we do not need checks for short circuit operations or other operations with a defined order of evaluation. The rules about arbitrary order (see 1.1.4) allow evaluating parameters and writing parameters back in an arbitrary order, but not interleaving of evaluating parameters of one call with writing parameters back from another — that would not correspond to any allowed sequential order.

6.20/3 {*AI05-0144-2*} For the purposes of checking this rule:

6.21/3 - For an array aggregate, an expression associated with a discrete_choice_list that has two or more discrete choices, or that has a nonstatic range, is considered as two or more separate occurrences of the expression;

6.22/3 - For a record aggregate:

6.23/3 - The expression of a record_component_association is considered to occur once for each associated component; and

6.24/3 - The default_expression for each record_component_association with <> for which the associated component has a default_expression is considered part of the aggregate;

6.25/3 - For a call, any default_expression evaluated as part of the call is considered part of the call.

6.aa/3 **Ramification:** We do not check expressions that are evaluated only because of a component initialized by default in an aggregate (via <>).

Dynamic Semantics

7 For the evaluation of a parameter_association:

8 - The actual parameter is first evaluated.

- For an access parameter, the access_definition is elaborated, which creates the anonymous access type. 9
- For a parameter [(of any mode)] that is passed by reference (see 6.2), a view conversion of the actual parameter to the nominal subtype of the formal parameter is evaluated, and the formal parameter denotes that conversion. 10

  Discussion: We are always allowing sliding, even for [**in**] **out** by-reference parameters. 10.a
- For an **in** or **in out** parameter that is passed by copy (see 6.2), the formal parameter object is created, and the value of the actual parameter is converted to the nominal subtype of the formal parameter and assigned to the formal. 11

  Ramification: The conversion mentioned here is a value conversion. 11.a
- For an **out** parameter that is passed by copy, the formal parameter object is created, and: 12
  - {*AI05-0153-3*} {*AI05-0196-1*} {*AI12-0378-1*} For an access type, the formal parameter is initialized from the value of the actual, without checking whether the value satisfies any constraints, predicates, or null exclusions, but including any[dynamic] accessibility checks associated with a conversion to the type of the formal parameter. 13/5

    Reason: This preserves the Language Design Principle that an object of an access type is always initialized with a "reasonable" value. 13.a

    Ramification: {*AI12-0378-1*} The permission to pass **null** (see below) can be used in any case where an accessibility check could fail, rather than making a check. 13.b/5
  - {*AI05-0153-3*} {*AI05-0228-1*} {*AI12-0074-1*} {*AI12-0159-1*} {*AI12-0377-1*} For a scalar type that has the Default_Value aspect specified, the formal parameter is initialized from the value of the actual, without checking that the value satisfies any constraint or any predicate. 13.1/5

    Reason: This preserves the Language Design Principle that all objects of a type with an implicit initial value are initialized. This is important so that a programmer can guarantee that all objects of a scalar type have a valid value with a carefully chosen Default_Value. 13.c/3

    Implementation Note: This rule means that **out** parameters of a subtype *T* with a specified Default_Value need to be large enough to support any possible value of the base type of *T*. In contrast, a type that does not have a Default_Value only need support the size of the subtype (since no values are passed in). 13.d/3
  - {*AI12-0333-1*} For a composite type with discriminants or that has implicit initial values for any subcomponents (see 3.3.1), the behavior is as for an **in out** parameter passed by copy[, except that no predicate check is performed]. 14/5

    Reason: This ensures that no part of an object of such a type can become "de-initialized" by being part of an **out** parameter. 14.a

    Ramification: This includes an array type whose component type is an access type, and a record type with a component that has a default_expression, among other things. 14.b

    Proof: {*AI12-0333-1*} No predicate check follows from the definition of subtype conversion in 4.6. 14.c/5
  - {*AI12-0439-1*} For any other type, the formal parameter is uninitialized. If composite, a view conversion of the actual parameter to the nominal subtype of the formal is evaluated [(which can raise Constraint_Error)], and the actual subtype of the formal is that of the view conversion. If elementary, the actual subtype of the formal is given by its nominal subtype. 15/5

    Ramification: {*AI05-0228-1*} This case covers scalar types that do not have Default_Value specified, and composite types whose subcomponent's subtypes do not have any implicit initial values. The view conversion for composite types ensures that if the lengths don't match between an actual and a formal array parameter, the Constraint_Error is raised before the call, rather than after. 15.a/3
  - {*AI12-0377-1*} Furthermore, if the type is a scalar type, and the actual parameter is a view conversion, then Program_Error is raised if either the target or the operand type has the Default_Value aspect specified, unless they both have the Default_Value aspect specified, and there is a type (other than a root numeric type) that is an ancestor of both the target type and the operand type. 15.1/5

15.b/5 **Discussion:** This can only occur in the body of an instance of a generic unit. Legality Rules will catch all other cases. Implementations that macro-expand generics can always detect this case when the enclosing instance body is expanded.

15.2/5 - {*AI05-0142-4*} {*AI05-0234-1*} In a function call, for each explicitly aliased parameter, a check is made that the accessibility level of the master of the actual object is not deeper than that of the master of the call (see 3.10.2).

15.c/3 **Ramification:** If the actual object to a call *C* is a formal parameter of some function call *F*, no dynamic check against the master of the actual parameter of *F* is necessary. Any case which could fail the dynamic check is already statically illegal (either at the call site of *F*, or at the call site *C*). This is important, as it would require nasty distributed overhead to accurately know the dynamic accessibility of a formal parameter (all tagged and explicitly aliased parameters would have to carry accessibility levels).

16 A formal parameter of mode **in out** or **out** with discriminants is constrained if either its nominal subtype or the actual parameter is constrained.

17 After normal completion and leaving of a subprogram, for each **in out** or **out** parameter that is passed by copy, the value of the formal parameter is converted to the subtype of the variable given as the actual parameter and assigned to it. These conversions and assignments occur in an arbitrary order.

17.a **Ramification:** The conversions mentioned above during parameter passing might raise Constraint_Error — (see 4.6).

17.b **Ramification:** If any conversion or assignment as part of parameter passing propagates an exception, the exception is raised at the place of the subprogram call; that is, it cannot be handled inside the subprogram_body.

17.c **Proof:** Since these checks happen before or after executing the subprogram_body, the execution of the subprogram_body does not dynamically enclose them, so it can't handle the exceptions.

17.d **Discussion:** The variable we're talking about is the one denoted by the *variable*_name given as the explicit_actual_parameter. If this *variable*_name is a type_conversion, then the rules in 4.6 for assigning to a view conversion apply. That is, if X is of subtype S1, and the actual is S2(X), the above-mentioned conversion will convert to S2, and the one mentioned in 4.6 will convert to S1.

Erroneous Execution

18/3 {*AI05-0008-1*} If the nominal subtype of a formal parameter with discriminants is constrained or indefinite, and the parameter is passed by reference, then the execution of the call is erroneous if the value of any discriminant of the actual is changed while the formal parameter exists (that is, before leaving the corresponding callable construct).

Implementation Permissions

19/5 {*AI12-0378-1*} If the actual parameter in a parameter_association with mode **out** is a view conversion between two access types that do not share a common ancestor type, the implementation may pass in the null value of the type of the formal parameter instead of the value of the actual parameter. It is implementation-defined under what circumstances the implementation passes in the null value.

19.a/5 **Implementation defined:** The circumstances in which the implementation passes in the null value for a view conversion of an access type used as an **out** parameter.

Extensions to Ada 83

19.b In Ada 95, a program can rely on the fact that passing an object as an **out** parameter does not "de-initialize" any parts of the object whose subtypes have implicit initial values. (This generalizes the RM83 rule that required copy-in for parts that were discriminants or of an access type.)

Wording Changes from Ada 83

19.c/3 {*AI05-0299-1*} We have eliminated the subclause on Default Parameters, as it is subsumed by earlier subclauses.

Inconsistencies With Ada 2005

19.d/3 {*AI05-0196-1*} **Correction:** Clarified that **out** parameters of an access type are not checked for null exclusions when they are passed in (which is similar to the behavior for constraints). This was unspecified in Ada 2005, so a program

which depends on the behavior of an implementation which does check the exclusion may malfunction. But a program depending on an exception being raised is unlikely.

Incompatibilities With Ada 2005

{*AI05-0144-2*} Additional rules have been added to make illegal passing the same elementary object to more than one **in out** or **out** parameters of the same call. In this case, the result in the object could depend on the compiler version, optimization settings, and potentially the phase of the moon, so this check will mostly reject programs that are nonportable and could fail with any change. Even when the result is expected to be the same in both parameters, the code is unnecessarily tricky. Programs which fail this new check should be rare and are easily fixed by adding a temporary object. 19.e/3

Wording Changes from Ada 2005

{*AI05-0008-1*} **Correction:** A missing rule was added to cover cases that were missed in Ada 95 and Ada 2005; specifically, that an **in** parameter passed by reference might have its discriminants changed via another path. Such cases are erroneous as requiring compilers to detect such errors would be expensive, and requiring such cases to work would be a major change of the user model (**in** parameters with discriminants could no longer be assumed constant). This is not an inconsistency, as compilers are not required to change any current behavior. 19.f/3

{*AI05-0102-1*} **Correction:** Moved implicit conversion Legality Rule to 8.6. 19.g/3

{*AI05-0118-1*} **Correction:** Added a definition for positional parameters, as this is missing from Ada 95 and later. 19.h/3

{*AI05-0142-4*} Rules have been added defining the legality and dynamic checks needed for explicitly aliased parameters (see 6.1). 19.i/3

{*AI05-0144-2*} Additional rules have been added such that passing an object to an **in out** or **out** parameter of a function is illegal if it is used elsewhere in a construct which allows evaluation in an arbitrary order. Such calls are not portable (since the results may depend on the evaluation order), and the results could even vary because of optimization settings and the like. Thus they've been banned. 19.j/3

Inconsistencies With Ada 2012

{*AI12-0378-1*} **Correction:** Added a permission to pass **null** so that value passed into an **out** parameter for access types is well-defined in the case of a view conversion. **Null** may be passed for any view conversion between unrelated access types; this is important for conversions that may have problematic accessibility or tags. If the permission is used and the **out** parameter is read before it is written (perhaps to read a bound or discriminant), Constraint_Error may be raised by Ada 2022 when it would not have been in Ada 2012. Additionally, if the called subprogram does not write the **out** parameter at all, the actual object will be overwritten with **null** (and possibly raise Constraint_Error if the object is null excluding), while the object would be unchanged in Ada 2012. Such cases are thought to be rare, as most **out** parameters of access types are overwritten before being read. In addition, at least one widely-used Ada compiler already passes **null** in these cases. 19.k/5

Incompatibilities With Ada 2012

{*AI12-0074-1*} {*AI12-0159-1*} {*AI12-0377-1*} {*AI12-0378-1*} **Corrigendum:** Added rules to ensure that the value passed into an **out** parameter for scalar types is well-defined in the case of a view conversion. The new rules can be incompatible. View conversions from/to an unrelated type with the Default_Value aspect specified are unlikely to occur in existing code, as the aspect is new in Ada 2012. Declaring and passing a temporary rather than a view conversion will eliminate the problem. 19.l/5

{*AI12-0095-1*} **Corrigendum:** Because of a rule added in 12.5.1, the checks for the passing of an object to an explicitly aliased parameter in a generic body were strengthened to use an assume the worst rule. This case is rather unlikely as a formal private or derived type with discriminants is required along with an explicitly aliased parameter whose type doesn't statically match the formal type. Such a program is very unlikely, especially as explicitly aliased parameters are a new Ada 2012 feature. 19.m/4

Wording Changes from Ada 2012

{*AI12-0216-1*} **Correction:** The **in out** parameter rule only applies to actual parameters of elementary types. While this allows additional programs (and thus could be considered an extension), it is unlikely to change anything in a real program (it could only matter in a call with 4 or more parameters, and then only if two composite parameters have matching actuals). Thus we document it as a wording change. 19.n/5

{*AI12-0333-1*} **Correction:** Predicate checks are not made for inbound **out** parameters. The actual rule change that has this effect is found in 4.6 and is documented there. 19.o/5

6.5 Return Statements

1/2 {*AI95-00318-02*} A simple_return_statement or extended_return_statement (collectively called a *return statement*) is used to complete the execution of the innermost enclosing subprogram_body, entry_body, or accept_statement.

Syntax

2/2 {*AI95-00318-02*} simple_return_statement ::= **return** [expression];

2.1/5 {*AI05-0277-1*} {*AI12-0398-1*} {*AI12-0446-1*} extended_return_object_declaration ::=
defining_identifier : [**aliased**] [**constant**] return_subtype_indication [:= expression]
[aspect_specification]

2.2/3 {*AI95-00318-02*} {*AI05-0015-1*} {*AI05-0053-1*} {*AI05-0277-1*} {*AI05-0299-1*}
extended_return_statement ::=
return extended_return_object_declaration [**do**
handled_sequence_of_statements
end return];

2.3/2 {*AI95-00318-02*} return_subtype_indication ::= subtype_indication | access_definition

Name Resolution Rules

3/5 {*AI95-00318-02*} {*AI12-0173-1*} The *result subtype* of a function is the subtype denoted by the subtype_mark, or defined by the access_definition, after the reserved word **return** in the profile of the function. The expected type for the expression, if any, of a simple_return_statement is the result type of the corresponding function. The expected type for the expression of an extended_return_object_declaration is that of the return_subtype_indication.

3.a **To be honest:** The same applies to generic functions.

Legality Rules

4/2 {*AI95-00318-02*} A return statement shall be within a callable construct, and it *applies to* the innermost callable construct or extended_return_statement that contains it. A return statement shall not be within a body that is within the construct to which the return statement applies.

4.a/4 **To be honest:** {*AI12-0089-1*} The above also applies to generic subprograms, even though they are not callable constructs. (An instance of a generic subprogram is a callable construct, but not a generic subprogram itself.)

5/5 {*AI95-00318-02*} {*AI05-0015-1*} {*AI12-0173-1*} A function body shall contain at least one return statement that applies to the function body, unless the function contains code_statements. A simple_-return_statement shall include an expression if and only if it applies to a function body. An extended_return_statement shall apply to a function body. An extended_return_object_declaration with the reserved word **constant** shall include an expression.

5.a/4 **Reason:** {*AI95-00318-02*} {*AI12-0022-1*} The requirement that a function body has to have at least one return statement is a "helpful" restriction. There has been some interest in lifting this restriction, or allowing a raise statement to substitute for the return statement. However, there was enough interest in leaving it as is that we decided not to change it. Note that for Ada 2012, Corrigendum 1, a return statement whose expression is a raise_expression can be given in *any* function body (the raise_expression will match any type), so there is much less need to eliminate this rule.

5.b/2 **Ramification:** {*AI95-00318-02*} A return statement can apply to an extended_return_statement, so a simple_-return_statement without an expression can be given in one. However, neither simple_return_statement with an expression nor an extended_return_statement can be given inside an extended_return_statement, as they must apply (directly) to a function body.

{*AI12-0089-1*} Since a "function body" includes a generic function body, this rule and all of the following Legality Rules apply to generic function bodies as well as non-generic function bodies. This is true even though a generic function is not a function. 5.b.1/4

{*AI12-0173-1*} {*AI12-0418-1*} The *expression of an extended_return_statement* is the expression (if any) of the extended_return_object_declaration of the extended_return_statement. 5.1/5

Ramification: {*AI12-0173-1*} {*AI12-0418-1*} The *expression of a return statement* is either the expression of a simple_return_statement or the expression of an extended_return_statement as defined above. 5.b.2/5

{*AI95-00318-02*} For an extended_return_statement that applies to a function body: 5.2/2

- {*AI95-00318-02*} {*AI05-0032-1*} {*AI05-0103-1*} If the result subtype of the function is defined by a subtype_mark, the return_subtype_indication shall be a subtype_indication. The type of the subtype_indication shall be covered by the result type of the function. The subtype defined by the subtype_indication shall be statically compatible with the result subtype of the function; if the result type of the function is elementary, the two subtypes shall statically match. If the result subtype of the function is indefinite, then the subtype defined by the subtype_indication shall be a definite subtype, or there shall be an expression. 5.3/5
- {*AI95-00318-02*} If the result subtype of the function is defined by an access_definition, the return_subtype_indication shall be an access_definition. The subtype defined by the access_definition shall statically match the result subtype of the function. [The accessibility level of this anonymous access subtype is that of the result subtype.] 5.4/5

  Proof: {*AI12-0070-1*} The accessibility of such anonymous access types is defined in the Heart of Darkness (aka 3.10.2). 5.b.3/4
- {*AI05-0032-1*} If the result subtype of the function is class-wide, the accessibility level of the type of the subtype defined by the return_subtype_indication shall not be statically deeper than that of the master that elaborated the function body. 5.5/5

  Reason: In this case, the return_subtype_indication could be a specific type initialized by default; in that case there is no expression to check. 5.b.4/3

{*AI95-00318-02*} {*AI05-0032-1*} For any return statement that applies to a function body: 5.6/5

- {*AI95-00318-02*} {*AI05-0188-1*} [If the result subtype of the function is limited, then the expression of the return statement (if any) shall meet the restrictions described in 7.5.] 5.7/5

  This paragraph was deleted.{*AI05-0188-1*} 5.c/3
- {*AI95-00416-01*} {*AI05-0032-1*} {*AI05-0051-1*} If the result subtype of the function is class-wide, the accessibility level of the type of the expression (if any) of the return statement shall not be statically deeper than that of the master that elaborated the function body. 5.8/5

  Discussion: {*AI05-0032-1*} {*AI05-0051-1*} {*AI12-0005-1*} If the result type is class-wide, then there must be an expression of the return statement unless this is an extended_return_statement whose return_subtype_indication is a specific type. We have a separate rule to cover that case. Note that if an extended_return_statement has an expression, then both this rule and the next one must be satisfied. 5.d/4
- {*AI05-0051-1*} If the subtype determined by the expression of the simple_return_statement or by the return_subtype_indication has one or more access discriminants, the accessibility level of the anonymous access type of each access discriminant shall not be statically deeper than that of the master that elaborated the function body. 5.9/5

  Discussion: We use the type used by the return statement rather than from the function return type since we want to check whenever the return object has access discriminants, even if the function return type doesn't have any (mostly for a class-wide type). 5.d.1/3

{*AI05-0277-1*} {*AI12-0426-1*} If the reserved word **aliased** is present in an extended_return_object_declaration, the type of the extended return object shall be immutably limited. 5.10/5

Static Semantics

5.11/5 {*AI95-00318-02*} {*AI05-0015-1*} {*AI05-0144-2*} Within an extended_return_statement, the *return object* is declared with the given defining_identifier, with the nominal subtype defined by the return_subtype_indication. An extended_return_statement with the reserved word **constant** is a full constant declaration that declares the return object to be a constant object.

Dynamic Semantics

5.12/5 {*AI95-00318-02*} {*AI95-00416-01*} {*AI05-0032-1*} {*AI12-0439-1*} For the execution of an extended_return_statement, the subtype_indication or access_definition is elaborated. This creates the nominal subtype of the return object. If there is an expression, it is evaluated and converted to the nominal subtype (which can raise Constraint_Error — see 4.6); the return object is created and the converted value is assigned to the return object. Otherwise, the return object is created and initialized by default as for a stand-alone object of its nominal subtype (see 3.3.1). If the nominal subtype is indefinite, the return object is constrained by its initial value. A check is made that the value of the return object belongs to the function result subtype. Constraint_Error is raised if this check fails.

5.e/2 **Ramification:** If the result type is controlled or has a controlled part, appropriate calls on Initialize or Adjust are performed prior to executing the handled_sequence_of_statements, except when the initial expression is an aggregate (which requires build-in-place with no call on Adjust).

5.f/3 {*AI05-0005-1*} If the return statement is left without resulting in a return (for example, due to an exception propagated from the expression or the handled_sequence_of_statements, or a goto out of the handled_sequence_of_statements), if the return object has been created, it is finalized prior to leaving the return statement. If it has not been created when the return statement is left, it is not created or finalized.

5.g/3 {*AI05-0032-1*} Other rules ensure that the check required by this rule cannot fail unless the function has a class-wide result subtype where the associated specific subtype is constrained. In other cases, either the subtypes have to match or the function's subtype is unconstrained and needs no checking.

6/2 {*AI95-00318-02*} For the execution of a simple_return_statement, the expression (if any) is first evaluated, converted to the result subtype, and then is assigned to the anonymous *return object*.

6.a **Ramification:** The conversion might raise Constraint_Error — (see 4.6).

7/2 {*AI95-00318-02*} {*AI95-00416-01*} [If the return object has any parts that are tasks, the activation of those tasks does not occur until after the function returns (see 9.2).]

7.a/2 **Proof:** This is specified by the rules in 9.2.

7.b/2 **Reason:** Only the caller can know when task activations should take place, as it depends on the context of the call. If the function is being used to initialize the component of some larger object, then that entire object must be initialized before any task activations. Even after the outer object is fully initialized, task activations are still postponed until the **begin** at the end of the declarative part if the function is being used to initialize part of a declared object.

8/5 {*AI95-00318-02*} {*AI95-00344-01*} {*AI05-0024-1*} {*AI05-0032-1*} {*AI12-0097-1*} {*AI12-0418-1*} If the result type of a function is a specific tagged type, the tag of the return object is that of the result type. If the result type is class-wide, the tag of the return object is that of the value of the expression of the return statement, unless the return object is defined by an extended_return_object_declaration with a subtype_indication that is specific, in which case it is that of the type of the subtype_indication. A check is made that the master of the type identified by the tag of the result includes the elaboration of the master that elaborated the function body. If this check fails, Program_Error is raised.

8.a/2 **Ramification:** {*AI95-00318-02*} The first sentence is true even if the tag of the expression is different, which could happen if the expression were a view conversion or a dereference of an access value. Note that for a limited type, because of the restriction to aggregates and function calls (and no conversions), the tag will already match.

8.b/2 **Reason:** {*AI95-00318-02*} The first rule ensures that a function whose result type is a specific tagged type always returns an object whose tag is that of the result type. This is important for dispatching on controlling result, and allows the caller to allocate the appropriate amount of space to hold the value being returned (assuming there are no discriminants).

The master check prevents the returned object from outliving its type. Note that this check cannot fail for a specific tagged type, as the tag represents the function's type, which necessarily must be declared outside of the function. 8.c/3

We can't use the normal accessibility level "deeper than" check here because we may have "incomparable" levels if the masters belong to two different tasks. This can happen when an accept statement calls a function declared in the enclosing task body, and the function returns an object passed to it from the accept statement, and this object was itself a parameter to the accept statement. 8.d/3

{*AI95-00318-02*} {*AI05-0058-1*} {*AI12-0343-1*} For the execution of an extended_return_statement, the handled_sequence_of_statements is executed. Within this handled_sequence_of_statements, the execution of a simple_return_statement that applies to the extended_return_statement causes a transfer of control that completes the extended_return_statement. Upon completion of a return statement that applies to a callable construct by the normal completion of a simple_return_statement or by reaching the **end return** of an extended_return_statement, a transfer of control is performed which completes the execution of the callable construct, and returns to the caller. 8.1/5

Ramification: {*AI05-0058-1*} A transfer of control that completes an extended_return_statement (such as an exit or goto) does not cause a return to the caller unless it is caused by simple_return_statement (that is, triggers the second sentence of this paragraph). The return to the caller occurs for the simple_return_statement that applies to an extended_return_statement because the last sentence says "the normal completion of a simple_return_statement", which includes the one nested in the extended_return_statement. 8.e/5

{*AI12-0343-1*} The check on the tag of the object occurs when the object is created (before any sequence_of_statements); the checks which follow occur after the execution of any sequence_of_statements. This is implicit in the order of definition of these Dynamic Semantics. 8.f/5

{*AI05-0073-1*} {*AI12-0343-1*} If the result subtype of the function is defined by an access_definition designating a specific tagged type *T*, a check is made that the result value is null or the tag of the object designated by the result value identifies *T*. Constraint_Error is raised if this check fails. 8.2/5

Reason: This check is needed so that dispatching on controlling access results works for tag-indeterminate functions. If it was not made, it would be possible for such functions to return an access to a descendant type, meaning the function could return an object with a tag different than the one assumed by the dispatching rules. 8.g/3

Paragraphs 9 through 20 were deleted.

{*AI95-00318-02*} {*AI95-00402-01*} {*AI95-00416-01*} {*AI05-0051-1*} If any part of the specific type of the return object of a function (or coextension thereof) has one or more access discriminants whose value is not constrained by the result subtype of the function, a check is made that the accessibility level of the anonymous access type of each access discriminant, as determined by the expression or the return_subtype_indication of the return statement, is not deeper than the level of the master of the call (see 3.10.2). If this check fails, Program_Error is raised. 21/3

This paragraph was deleted. 21.a/2

Reason: The check prevents the returned object (for a nonlimited type) from outliving the object designated by one of its discriminants. The check is made on the values of the discriminants, which may come from the return_subtype_indication (if constrained), or the expression, but it is never necessary to check both. 21.b/2

Implementation Note: {*AI05-0234-1*} The reason for saying "any part of the specific type" is to simplify implementation. In the case of class-wide result objects, this allows the testing of a simple flag in the tagged type descriptor that indicates whether the specific type has any parts with access discriminants. By basing the test on the type of the object rather than the object itself, we avoid concerns about whether subcomponents in variant parts and of arrays (which might be empty) are present. 21.c/3

Discussion: {*AI05-0234-1*} For a function with a class-wide result type, the access values that need to be checked are determined by the tag of the return object. In order to implement this accessibility check in the case where the tag of the result is not known statically at the point of the return statement, an implementation may need to somehow associate with the tag of a specific tagged type an indication of whether the type has unconstrained access discriminants (explicit or inherited) or has any subcomponents with such discriminants. If an implementation is already maintaining a statically initialized descriptor of some kind for each specific tagged type, then an additional Boolean could be added to this descriptor. 21.d/3

{*AI05-0005-1*} {*AI05-0234-1*} Note that the flag should only be queried in the case where the result object might have access discriminants that might have subtypes with "bad" accessibility levels (as determined by the rules of 3.10.2 for 21.e/3

determining the accessibility level of the type of an access discriminant in the expression or return_subtype_indication of a return statement).

21.f/3 Thus, in a case like

21.g/3
```
type Global is access T'Class;
function F (Ptr : Global) return T'Class is
begin
   return Ptr.all;
end F;
```

21.h/3 there is no need for a run-time accessibility check. While an object of T'Class "might have" access discriminants, the accessibility of those potential discriminants cannot be bad. The setting of the bit doesn't matter and there is no need to query it.

21.i/3 On the other hand, given

21.j/3
```
function F return T'Class is
   Local : T'Class := ... ;
begin
   return Local;
end F;
```

21.k/3 In this case, a check would typically be required.

21.l/3 The need for including subcomponents in this check is illustrated by the following example:

21.m/3
```
X : aliased Integer;
```

21.n/3
```
type Component_Type (Discrim : access Integer := X'Access)
   is limited null record;
```

21.o/3
```
type Undiscriminated is record
   Fld : Component_Type;
end record;
```

21.p/3
```
function F return Undiscriminated is
   Local : aliased Integer;
begin
   return X : Undiscriminated := (Fld => (Discrim => Local'Access)) do
      Foo;
   end return;
   -- raises Program_Error after calling Foo.
end F;
```

21.q/3 **Ramification:** {*AI05-0234-1*} In the case where the tag of the result is not known statically at the point of the return statement and the run-time accessibility check is needed, discriminant values and array bounds play no role in performing this check. That is, array components are assumed to have nonzero length and components declared within variant parts are assumed to be present. Thus, the check may be implemented simply by testing the aforementioned descriptor bit and conditionally raising Program_Error.

22/5 {*AI95-00318-02*} {*AI05-0058-1*} {*AI12-0343-1*} {*AI12-0449-1*} A check is performed that the return value satisfies the predicates of the return subtype. If this check fails, the effect is as defined in 3.2.4.

22.a/5 *This paragraph was deleted.*{*AI05-0058-1*} {*AI12-0343-1*}

22.b/5 **Implementation Note:** {*AI12-0005-1*} {*AI12-0343-1*} The subtype conversion of the return expression for a simple_return_statement performs this same check. The permissions of 11.4.2 ensure that duplicate evaluation of a predicate at a single point is not required (other than pathological, not portable cases), so a single evaluation of the predicate is enough in this case.

23/2 {*AI95-00318-02*} In the case of a function, the function_call denotes a constant view of the return object.

Implementation Permissions

24/3 {*AI95-00416-01*} {*AI05-0050-1*} For a function call used to initialize a composite object with a constrained nominal subtype or used to initialize a return object that is built in place into such an object:

24.1/3 - {*AI05-0050-1*} If the result subtype of the function is constrained, and conversion of an object of this subtype to the subtype of the object being initialized would raise Constraint_Error, then Constraint_Error may be raised before calling the function.

24.2/3 - {*AI05-0050-1*} If the result subtype of the function is unconstrained, and a return statement is executed such that the return object is known to be constrained, and conversion of the return object to the subtype of the object being initialized would raise Constraint_Error, then

Constraint_Error may be raised at the point of the call (after abandoning the execution of the function body).

Reason: {*AI95-00416-01*} {*AI05-0050-1*} Without such a permission, it would be very difficult to implement "built-in-place" semantics. The intention is that the exception is raised at the same point that it would have been raised without the permission; it should not change handlers if the implementation switches between return-by-copy and built-in-place. This means that the exception is not handleable within the function, because in the return-by-copy case, the constraint check to verify that the result satisfies the constraints of the object being initialized happens after the function returns. This implies further that upon detecting such a situation, the implementation may need to simulate a goto to a point outside any local exception handlers prior to raising the exception. 24.a/3

Ramification: {*AI95-00416-01*} {*AI05-0050-1*} These permissions do not apply in the case of an extended return object with mutable discriminants. That's necessary because in that case a return object can be created with the "wrong" discriminants and then changed to the "right" discriminants later (but before returning). We don't want this case raising an exception when the canonical semantics will not do so. 24.b/3

{*AI05-0050-1*} It's still possible to write a program that will raise an exception using this permission that would not in the canonical semantics. That could happen if a return statement with the "wrong" discriminants or bounds is abandoned (via an exception, or for an extended_return_statement, via an exit or goto statement), and then a return statement with the "right" discriminants or bounds is executed. The only solution for this problem is to not have the permission at all, but this is too unusual of a case to worry about the effects of the permission, especially given the implementation difficulties for built-in-place objects that this permission is intended to ease. 24.c/3

{*AI05-0050-1*} Note that the mutable-discriminant case only happens when built-in-place initialization is optional. This means that any difficulties associated with implementing built-in-place initialization without these permissions can be sidestepped by not building in place. 24.d/3

Examples

Examples of return statements: 25

```
{AI95-00318-02} return;                          -- in a procedure body, entry_body,                    26/2
                                                 -- accept_statement, or extended_return_statement
return Key_Value(Last_Index);                    -- in a function body                                  27
{AI95-00318-02} return Node : Cell do            -- in a function body, see 3.10.1 for Cell             28/2
   Node.Value := Result;
   Node.Succ := Next_Node;
end return;
```

Incompatibilities With Ada 83

{*AI95-00318-02*} In Ada 95, if the result type of a function has a part that is a task, then an attempt to return a local variable will raise Program_Error. This is illegal in Ada 2005, see below. In Ada 83, if a function returns a local variable containing a task, execution is erroneous according to AI83-00867. However, there are other situations where functions that return tasks (or that return a variant record only one of whose variants includes a task) are correct in Ada 83 but will raise Program_Error according to the new rules. 28.a/2

The rule change was made because there will be more types (protected types, limited controlled types) in Ada 95 for which it will be meaningless to return a local variable, and making all of these erroneous is unacceptable. The current rule was felt to be the simplest that kept upward incompatibilities to situations involving returning tasks, which are quite rare. 28.b

Wording Changes from Ada 83

{*AI05-0299-1*} This subclause has been moved here from chapter 5, since it has mainly to do with subprograms. 28.c/3

A function now creates an anonymous object. This is necessary so that controlled types will work. 28.d

{*AI95-00318-02*} We have clarified that a return statement applies to a callable construct, not to a callable entity. 28.e/2

{*AI95-00318-02*} There is no need to mention generics in the rules about where a return statement can appear and what it applies to; the phrase "body of a subprogram or generic subprogram" is syntactic, and refers exactly to "subprogram_body". 28.f/2

Inconsistencies With Ada 95

{*AI95-0416-1*} {*AI05-0005-1*} {*AI05-0050-1*} Added an Implementation Permission allowing early raising of Constraint_Error if the result cannot fit in the ultimate object. This gives implementations more flexibility to do built-in-place returns, and is essential for limited types (which cannot be built in a temporary). However, it allows raising 28.f.1/3

Constraint_Error in some cases where it would not be raised if the permission was not used. See Inconsistencies With Ada 2005 for additional changes. This case is potentially inconsistent with Ada 95, but a compiler does not have to take advantage of these permissions for any Ada 95 code, so there should be little practical impact.

Incompatibilities With Ada 95

28.g/2 {*AI95-00318-02*} The entire business about return-by-reference types has been dropped. Instead, the expression of a return statement of a limited type can only be an aggregate or function_call (see 7.5). This means that returning a global object or type_conversion, legal in Ada 95, is now illegal. Such functions can be converted to use anonymous access return types by adding **access** in the function definition and return statement, adding .**all** in uses, and adding **aliased** in the object declarations. This has the advantage of making the reference return semantics much clearer to the casual reader.

28.h/2 We changed these rules so that functions, combined with the new rules for limited types (7.5), can be used as build-in-place constructors for limited types. This reduces the differences between limited and nonlimited types, which will make limited types useful in more circumstances.

Extensions to Ada 95

28.i/2 {*AI95-00318-02*} The extended_return_statement is new. This provides a name for the object being returned, which reduces the copying needed to return complex objects (including no copying at all for limited objects). It also allows component-by-component construction of the return object.

Wording Changes from Ada 95

28.j/2 {*AI95-00318-02*} The wording was updated to support anonymous access return subtypes.

28.k/2 {*AI95-00318-02*} The term "return expression" was dropped because reviewers found it confusing when applied to the default expression of an extended_return_statement.

28.l/2 {*AI95-00344-01*} {*AI95-00416-01*} Added accessibility checks to class-wide return statements. These checks could not fail in Ada 95 (as all of the types had to be declared at the same level, so the tagged type would necessarily have been at the same level as the type of the object).

28.m/2 {*AI95-00402-01*} {*AI95-00416-01*} Added accessibility checks to return statements for types with access discriminants. Since such types have to be limited in Ada 95, the expression of a return statement would have been illegal in order for this check to fail.

Inconsistencies With Ada 2005

28.n/3 {*AI05-0050-1*} **Correction:** The Implementation Permission allowing early raising of Constraint_Error was modified to remove the most common of these cases from the permission (returning an object with mutable discriminants, where the return object is created with one set of discriminants and then changed to another). (The permission was also widened to allow the early check for constrained functions when that constraint is wrong.) However, there still is an unlikely case where the permission would allow an exception to be raised when none would be raised by the canonical semantics (when a return statement is abandoned). These changes can only remove the raising of an exception (or change the place where it is raised) compared to Ada 2005, so programs that depend on the previous behavior should be very rare.

28.o/3 {*AI05-0051-1*} {*AI05-0234-1*} **Correction:** Accessibility checks for access discriminants now depend on the master of the call rather than the point of declaration of the function. This will result in cases that used to raise Program_Error now running without raising any exception. This is technically inconsistent with Ada 2005 (as defined by Amendment 1), but it is unlikely that any real code depends on the raising of this exception.

28.p/3 {*AI05-0073-1*} **Correction:** Added a tag check for functions returning anonymous access-to-tagged types, so that dispatching of tag-indeterminate function works as expected. This is technically inconsistent with Ada 2005 (as defined by Amendment 1), but as the feature in question was newly added to Ada 2005, there should be little code that depends on the behavior that now raises an exception.

Incompatibilities With Ada 2005

28.q/3 {*AI05-0053-1*} {*AI05-0277-1*} **Correction:** The **aliased** keyword can now only appear on extended return objects with an immutably limited type. Other types would provide a way to get an aliased view of an object that is not necessarily aliased, which would be very bad. This is incompatible, but since the feature was added in Ada 2005, the keyword had no defined meaning in Ada 2005 (a significant oversight), and most sensible uses involve immutably limited types, it is unlikely that it appears meaningfully in existing programs.

28.r/3 {*AI05-0103-1*} **Correction:** Added wording to require static matching for unconstrained access types in extended return statements. This disallows adding or omitting null exclusions, and adding access constraints, in the declaration of the return object. While this is incompatible, the incompatible cases in question are either useless (access constraints

– the constraint can be given on an allocator if necessary, and still must be given there even if given on the return object) or wrong (null exclusions – null could be returned from a function declared to be null excluding), so we expect them to be extremely rare in practice.

Extensions to Ada 2005

{*AI05-0015-1*} {*AI05-0144-2*} The return object of an extended_return_statement can be declared constant; this works similarly to a constant object declaration. 28.s/3

{*AI05-0032-1*} Added wording to allow the return_subtype_indication to have a specific type if the return subtype of the function is class-wide. Specifying the (specific) type of the return object is awkward without this change, and this is consistent with the way allocators work. 28.t/3

Wording Changes from Ada 2005

{*AI05-0024-1*} **Correction:** Corrected the master check for tags since the masters may be for different tasks and thus incomparable. 28.u/3

{*AI05-0058-1*} **Correction:** Corrected the wording defining returns for extended_return_statements, since leaving by an exit or goto is considered “normal” completion of the statement. 28.v/3

{*AI05-0205-1*} {*AI05-0277-1*} **Correction:** Added the extended_return_object_declaration to make other rules easier to write and eliminate the problem described in AI05-0205-1. 28.w/3

Inconsistencies With Ada 2012

{*AI12-0343-1*} **Correction:** Reordered the checks associated with return statements to clarify when they happen relative to the sequence_of_statements of an extended_return_statement, and to add an additional predicate check on the returned value. This latter check can fail if the value to be returned is changed in the statements of an extended return. Thus, the following function now raises Assertion_Error, while no exception would be raised in Ada 2012: 28.x/5

```
type Rec is record                                                    28.y/5
  Count, Max : Natural;
end record with Dynamic_Predicate => Rec.Count <= Rec.Max;

function Foo return Rec is                                            28.z/5
begin
    return Result : Rec := (Count => 5, Max => 10) do
       Result.Max := 0;
    end return;
end Foo;
```

This will detect a program bug (returning a value that does not meet the predicate of the return subtype) rather than leaving it hidden, so the inconsistency is tolerable. 28.aa/5

Extensions to Ada 2012

{*AI12-0398-1*} Extended return objects now can have an aspect_specification, allowing the specification of (implementation-defined) aspects for return objects. 28.bb/5

Wording Changes from Ada 2012

{*AI05-0097-1*} **Corrigendum:** Clarified the wording so that it is clear where the tag of the return object comes from. While a literal reading of the original Ada 2012 rule could have caused some weird results (by using some nearby subtype_indication to provide the tag in the case of a simple_return_statement), such a reading would be so unlike the rest of the language that we do not believe anyone would ever have thought it was intended. As such, we do not believe any implementation ever did this wrong (at least because of the old wording), and thus do not document this as a possible inconsistency. 28.cc/4

{*AI12-0173-1*} **Correction:** Added a definition of the term expression of an extended_return_statement. That wording was commonly used before the extended_return_object_declaration was split into a separate syntax production, leaving the extended_return_statement without any expression of its own. Moreover, the wording often just uses “expression of a return statement” to cover both kinds of return statement. Changing the wording in more than a dozen places was unappealing, and some of the changes would be awkward to read, so we defined the term and left the majority of the wording unchanged. 28.dd/5

6.5.1 Nonreturning Subprograms

1/5 {*AI95-00329-01*} {*AI95-00414-01*} {*AI05-0229-1*} {*AI12-0269-1*} {*AI12-0418-1*} Specifying aspect No_Return to have the value True indicates that a subprogram cannot return normally[; it may, for example, propagate an exception or loop forever].

1.a/5 **Discussion:** Aspect No_Deposit will have to wait for Ada 2028. :-)

Paragraphs 2 and 3 were moved to Annex J, "Obsolescent Features".

Static Semantics

3.1/5 {*AI05-0229-1*} {*AI12-0269-1*} For a subprogram or generic subprogram, the following language-defined representation aspect may be specified:

3.2/3 No_Return The type of aspect No_Return is Boolean. When aspect No_Return is True for an entity, the entity is said to be *nonreturning*.

3.3/5 {*AI12-0423-1*} If directly specified, the aspect_definition shall be a static expression. When not directly specified, if the subprogram is a primitive subprogram inherited by a derived type, then the aspect is True if any corresponding subprogram of the parent or progenitor types is nonreturning. Otherwise, the aspect is False.

3.a/5 **Aspect Description for No_Return:** A subprogram will not return normally.

3.4/5 {*AI05-0229-1*} {*AI12-0269-1*} If a generic subprogram is nonreturning, then so are its instances. If a subprogram declared within a generic unit is nonreturning, then so are the corresponding copies of that subprogram in instances.

Legality Rules

4/3 {*AI95-00329-01*} {*AI95-00414-01*} {*AI05-0229-1*} Aspect No_Return shall not be specified for a null procedure nor an instance of a generic unit.

4.a/2 **Reason:** A null procedure cannot have the appropriate nonreturning semantics, as it does not raise an exception or loop forever.

4.b/5 **Ramification:** {*AI05-0229-1*} {*AI12-0269-1*} The procedure can be abstract. If a nonreturning subprogram is renamed (anywhere) calls through the new name still have the nonreturning semantics.

5/2 {*AI95-00329-01*} {*AI95-00414-01*} A return statement shall not apply to a nonreturning procedure or generic procedure.

5.1/5 {*AI12-0269-1*} Any return statement that applies to a nonreturning function or generic function shall be a simple_return_statement with an expression that is a raise_expression, a call on a nonreturning function, or a parenthesized expression of one of these.

5.a/5 **Ramification:** We still require at least one return statement in a function; we just require that all such return statements don't actually return a value.

6/5 {*AI95-00414-01*} {*AI12-0269-1*} A subprogram shall be nonreturning if it overrides a dispatching nonreturning subprogram. In addition to the places where Legality Rules normally apply (see 12.3), this rule applies also in the private part of an instance of a generic unit.

6.a/5 **Reason:** This ensures that dispatching calls to nonreturning subprograms will, in fact, not return.

7/5 {*AI95-00414-01*} {*AI12-0269-1*} If a renaming-as-body completes a nonreturning subprogram declaration, then the renamed subprogram shall be nonreturning.

7.a/2 **Reason:** This ensures that no extra code is needed to implement the renames (that is, no wrapper is needed) as the body has the same property.

Paragraph 8 was deleted.

Dynamic Semantics

{*AI95-00329-01*} {*AI95-00414-01*} If the body of a nonreturning procedure completes normally, Program_Error is raised at the point of the call. 9/2

Discussion: {*AI12-0269-1*} Note that there is no name for suppressing this check, since the check represents a bug, imposes no time overhead, and minimal space overhead (since it can usually be statically eliminated as dead code). We don't need a similar rule for nonreturning functions as this is standard semantics for all functions that normally complete without a transfer of control (such as a return statement). 9.a/5

Implementation Note: If a nonreturning procedure tries to return, we raise Program_Error. This is stated as happening at the call site, because we do not wish to allow the procedure to handle the exception (and then, perhaps, try to return again!). However, the expected run-time model is that the compiler will generate **raise** Program_Error at the end of the procedure body (but not handleable by the procedure itself), as opposed to doing it at the call site. (This is just like the typical run-time model for functions that fall off the end without returning a value). The reason is indirect calls: in P.**all**(...);, the compiler cannot know whether P designates a nonreturning procedure or a normal one. Putting the **raise** Program_Error in the procedure's generated code solves this problem neatly. 9.b/2

Similarly, if one passes a nonreturning procedure to a generic formal parameter, the compiler cannot know this at call sites (in shared code implementations); the raise-in-body solution deals with this neatly. 9.c/2

Examples

{*AI12-0429-1*} *Example of a specification of a No_Return aspect:* 10/5

```
{AI95-00433-01} {AI05-0229-1} procedure Fail(Msg : String)   -- raises Fatal_Error exception
   with No_Return;
   -- Inform compiler and reader that procedure never returns normally
```
11/3

Extensions to Ada 95

{*AI95-00329-01*} {*AI95-00414-01*} Pragma No_Return is new. 11.a/2

Extensions to Ada 2005

{*AI05-0229-1*} Aspect No_Return is new; pragma No_Return is now obsolescent. 11.b/3

Incompatibilities With Ada 2012

{*AI12-0423-1*} **Correction:** Clarified that the nonreturning property is inherited by derivation. A literal implementation of the Ada 2012 would mean that no overriding would ever be rejected, as the inherited routine would never be nonreturning. Ada 2022 requires overridings of dispatching nonreturning subprograms to be rejected. This is formally incompatible, but practically such overridings have been rejected in practice. 11.c/5

Extensions to Ada 2012

{*AI12-0269-1*} Aspect No_Return can now be applied to functions as well as procedures. 11.d/5

6.6 Overloading of Operators

An *operator* is a function whose designator is an operator_symbol. [Operators, like other functions, may be overloaded.] 1

Name Resolution Rules

Each use of a unary or binary operator is equivalent to a function_call with *function*_prefix being the corresponding operator_symbol, and with (respectively) one or two positional actual parameters being the operand(s) of the operator (in order). 2

To be honest: {*AI05-0299-1*} {*AI12-0449-1*} We also use the term operator (in Clause 4 and in 6.1) to refer to one of the syntactic categories defined in 4.5, “Operators and Expression Evaluation” whose names end with “_operator:” logical_operator, relational_operator, binary_adding_operator, unary_adding_operator, multiplying_operator, and highest_precedence_operator. 2.a/5

Discussion: {*AI05-0005-1*} This equivalence extends to uses of function_call in most other language rules. However, as often happens, the equivalence is not perfect, as operator calls are not a name, while a function_call is a name. Thus, operator calls cannot be used in contexts that require a name (such as a rename of an object). A direct fix for this 2.b/3

problem would be very disruptive, and thus we have not done that. However, qualifying an operator call can be used as a workaround in contexts that require a name.

Legality Rules

3/3 {*AI05-0143-1*} The subprogram_specification of a unary or binary operator shall have one or two parameters, respectively. The parameters shall be of mode **in**. A generic function instantiation whose designator is an operator_symbol is only allowed if the specification of the generic function has the corresponding number of parameters, and they are all of mode **in**.

4 Default_expressions are not allowed for the parameters of an operator (whether the operator is declared with an explicit subprogram_specification or by a generic_instantiation).

5 An explicit declaration of "/=" shall not have a result type of the predefined type Boolean.

Static Semantics

6/3 {*AI05-0128-1*} An explicit declaration of "=" whose result type is Boolean implicitly declares an operator "/=" that gives the complementary result.

6.a/3 **Discussion:** {*AI05-0128-1*} A "/=" defined by this rule is considered user-defined, which means that it will be inherited by a derived type. “User-defined” means “not language-defined” for the purposes of inheritance, that is anything other than predefined operators.

7/5 NOTE {*AI12-0440-1*} The operators "+" and "–" are both unary and binary operators, and hence can be overloaded with both one- and two-parameter functions.

Examples

8 *Examples of user-defined operators:*

9
```
function "+" (Left, Right : Matrix) return Matrix;
function "+" (Left, Right : Vector) return Vector;

-- assuming that A, B, and C are of the type Vector
-- the following two statements are equivalent:

A := B + C;
A := "+"(B, C);
```

Extensions to Ada 83

9.a Explicit declarations of "=" are now permitted for any combination of parameter and result types.

9.b Explicit declarations of "/=" are now permitted, so long as the result type is not Boolean.

Wording Changes from Ada 2005

9.c/3 {*AI05-0128-1*} **Correction:** Corrected the wording so that only explicit declarations of "=" cause an implicit declaration of "/="; otherwise, we could get multiple implicit definitions of "/=" without an obvious way to chose between them.

9.d/3 {*AI05-0143-1*} Added wording so that operators only allow parameters of mode **in**. This was made necessary by the elimination elsewhere of the restriction that function parameters be only of mode **in**.

6.7 Null Procedures

1/2 {*AI95-00348-01*} A null_procedure_declaration provides a shorthand to declare a procedure with an empty body.

Syntax

2/3 {*AI95-00348-01*} {*AI05-0183-1*} null_procedure_declaration ::=
[overriding_indicator]
procedure_specification **is null**

[aspect_specification];

Legality Rules

{*AI05-0177-1*} If a null_procedure_declaration is a completion, it shall be the completion of a subprogram_declaration or generic_subprogram_declaration. The profile of a null_procedure_declaration that completes a declaration shall conform fully to that of the declaration. 2.1/3

Static Semantics

{*AI95-00348-01*} {*AI05-0177-1*} {*AI05-0264-1*} {*AI12-0408-1*} A null_procedure_declaration that is not a completion declares a *null procedure*. A completion is not allowed for a null_procedure_declaration; however, a null_procedure_declaration can complete a previous declaration. 3/5

Reason: There are no null functions because the return value has to be constructed somehow; a function that always raises Program_Error doesn't seem very useful or worth the complication. 3.a/2

Dynamic Semantics

{*AI95-00348-01*} {*AI12-0408-1*} The execution of a null procedure is invoked by a subprogram call. For the execution of a subprogram call on a null procedure, or on a procedure completed with a null_procedure_declaration, the execution of the subprogram_body has no effect. 4/5

Ramification: Thus, a null procedure is equivalent to the body 4.a/2

```
begin
   null;
end;
```
4.b/2

with the exception that a null procedure can be used in place of a procedure specification. 4.c/2

{*AI95-00348-01*} {*AI05-0177-1*} The elaboration of a null_procedure_declaration has no other effect than to establish that the null procedure can be called without failing the Elaboration_Check. 5/3

Examples

{*AI12-0429-1*} *Example of the declaration of a null procedure:* 6/5

```
{AI95-00433-01} {AI12-0440-1} procedure Simplify(Expr : in out Expression) is null;
-- see 3.9
-- By default, Simplify does nothing, but it can be overridden in extensions of Expression
```
7/5

Extensions to Ada 95

{*AI95-00348-01*} Null procedures are new. 7.a/2

Extensions to Ada 2005

{*AI05-0177-1*} A null_procedure_declaration can now be a completion. 7.b/3

{*AI05-0183-1*} An optional aspect_specification can be used in a null_procedure_declaration. This is described in 13.1.1. 7.c/3

Wording Changes from Ada 2012

{*AI12-0408-1*} Clarified the term “null procedure” so it matches the meaning expected in 6.1. 7.d/5

6.8 Expression Functions

{*AI05-0177-1*} An expression_function_declaration provides a shorthand to declare a function whose body consists of a single return statement. 1/3

Syntax

2/4 {*AI95-0177-1*} {*AI12-0157-1*} expression_function_declaration ::=
[overriding_indicator]
function_specification **is**
(expression)
[aspect_specification];
| [overriding_indicator]
function_specification **is**
aggregate
[aspect_specification];

Name Resolution Rules

3/4 {*AI05-0177-1*} {*AI12-0157-1*} The expected type for the expression or aggregate of an expression_function_declaration is the result type (see 6.5) of the function.

Static Semantics

3.1/5 {*AI05-0177-1*} {*AI05-0264-1*} {*AI12-0075-1*} {*AI12-0157-1*} {*AI12-0408-1*} An expression_function_declaration that is not a completion declares an *expression function*. The *return expression of an expression function* is the expression or aggregate of the expression_function_declaration. A completion is not allowed for an expression_function_declaration; however, an expression_function_declaration can complete a previous declaration.

3.2/5 {*AI12-0075-1*} A *potentially static expression* is defined in the same way as a static expression except that

3.3/5 - a name denoting a formal parameter of an expression function is a potentially static expression; and

3.4/5 - each use of "static expression" in the definition of "static expression" is replaced with a corresponding use of "potentially static expression" in the definition of "potentially static expression".

3.a/5 **Discussion:** These uses occur in the definition of "static expression" in the cases of function calls, type conversions, qualified expressions, membership tests, short circuit control forms, conditional expressions, and parenthesized expressions.

3.5/5 {*AI12-0075-1*} The following language-defined representation aspect may be specified for an expression function:

3.6/5 Static The type of aspect Static is Boolean. When aspect Static is True for an expression function, the function is a *static expression function*. If directly specified, the aspect_definition shall be a static expression.

3.b/5 **Aspect Description for Static:** Specifies that an associated expression function can be used in static expressions.

3.7/5 The Static value for an inherited function is True if some corresponding primitive function of the parent or progenitor type is a static expression function; otherwise, if not directly specified, the aspect is False.

3.8/5 {*AI12-0075-1*} [A static expression function is a static function; see 4.9.]

Legality Rules

4/3 {*AI05-0177-1*} If an expression_function_declaration is a completion, it shall be the completion of a subprogram_declaration or generic_subprogram_declaration. The profile of an expression_function_declaration that completes a declaration shall conform fully to that of the declaration.

{*AI05-0177-1*} {*AI12-0157-1*} If the result subtype has one or more unconstrained access discriminants, the accessibility level of the anonymous access type of each access discriminant, as determined by the expression or aggregate of the expression_function_declaration, shall not be statically deeper than that of the master that elaborated the expression_function_declaration. 5/4

Ramification: This can only fail if the discriminant is an access to a part of a nonaliased parameter, as there can be no local declarations here. 5.a/3

Discussion: {*AI12-0005-1*} We don't need to repeat any of the other Legality Rules for return statements since none of them can fail here: the implicit return statement has to apply to this function (and isn't nested in something), there clearly is a return statement in this function, and the static class-wide accessibility check cannot fail as a tagged type cannot be declared locally in an expression function. 5.b/4

{*AI12-0075-1*} Aspect Static shall be specified to have the value True only if the associated expression_-function_declaration: 5.1/5

- is not a completion; 5.2/5
- has an expression that is a potentially static expression; 5.3/5
- contains no calls to itself; 5.4/5
- each parameter (if any) is of mode **in** and is of a static subtype; 5.5/5
- has a result subtype that is a static subtype; 5.6/5
- has no applicable precondition or postcondition expression; and 5.7/5
- {*AI12-0075-1*} {*AI12-0191-1*} for result type *R*, if the function is a boundary entity for type *R* (see 7.3.2), no type invariant applies to type *R*; if *R* has a component type *C*, a similar rule applies to *C*. 5.8/5

Ramification: Since a string subtype can be static, this allows an expression function of a string type to be static. 5.c/5

Paragraph 6 was deleted.

Dynamic Semantics

{*AI05-0177-1*} {*AI05-0262-1*} {*AI12-0157-1*} {*AI12-0408-1*} The execution of an expression function is invoked by a subprogram call. For the execution of a subprogram call on an expression function, or on a function completed with a expression_function_declaration, the execution of the subprogram_body executes an implicit function body containing only a simple_return_statement whose expression is the return expression of the expression function. 7/5

Discussion: The last sentence effectively means that all of the dynamic wording in 6.5 applies as needed, and we don't have to repeat it here. 7.a/3

{*AI05-0177-1*} The elaboration of an expression_function_declaration has no other effect than to establish that the expression function can be called without failing the Elaboration_Check. 8/3

Examples

{*AI12-0429-1*} *Example of an expression function:* 9/5

```
{AI05-0177-1} function Is_Origin (P : in Point) return Boolean is -- see 3.9
   (P.X = 0.0 and P.Y = 0.0);
```
10/3

Extensions to Ada 2005

{*AI05-0177-1*} Expression functions are new in Ada 2012. 10.a/3

Extensions to Ada 2012

{*AI12-0157-1*} **Corrigendum:** A aggregate can directly be the return expression of an expression function. This eliminates the double parentheses that otherwise would be necessary. 10.b/4

{*AI12-0075-1*} Aspect Static is new; it allows using suitable expression functions in static expressions. 10.c/5

Wording Changes from Ada 2012

10.d/5 {*AI12-0408-1*} Clarified the term “expression function” so it matches the meaning expected in 6.1.

7 Packages

[Packages are program units that allow the specification of groups of logically related entities. Typically, a package contains the declaration of a type (often a private type or private extension) along with the declarations of primitive subprograms of the type, which can be called from outside the package, while their inner workings remain hidden from outside users.] 1

Term entry: package — program unit that defines the interface to a group of logically related entities, along with their implementation 1.a/5
Note: Typically, a package contains the declaration of a type (often a private type or private extension) along with the declarations of primitive subprograms of the type, which can be called from outside the package, while their inner workings remain hidden from outside users.

7.1 Package Specifications and Declarations

[A package is generally provided in two parts: a package_specification and a package_body. Every package has a package_specification, but not all packages have a package_body.] 1

Syntax

package_declaration ::= package_specification; 2

{*AI05-0183-1*} package_specification ::= 3/3
package defining_program_unit_name
[aspect_specification] **is**
{basic_declarative_item}
[**private**
{basic_declarative_item}]
end [[parent_unit_name.]identifier]

If an identifier or parent_unit_name.identifier appears at the end of a package_specification, then this sequence of lexical elements shall repeat the defining_program_unit_name. 4

Legality Rules

{*AI95-00434-01*} A package_declaration or generic_package_declaration requires a completion [(a body)] if it contains any basic_declarative_item that requires a completion, but whose completion is not in its package_specification. 5/2

To be honest: {*AI05-0229-1*} If an implementation supports it, the body of a package or generic package may be imported (using aspect Import, see B.1), in which case no explicit body is allowed. 5.a/3

Static Semantics

{*AI95-00420-01*} {*AI95-00434-01*} The first list of basic_declarative_items of a package_specification of a package other than a generic formal package is called the *visible part* of the package. [The optional list of basic_declarative_items after the reserved word **private** (of any package_specification) is called the *private part* of the package. If the reserved word **private** does not appear, the package has an implicit empty private part.] Each list of basic_declarative_items of a package_specification forms a *declaration list* of the package. 6/2

Ramification: This definition of visible part does not apply to generic formal packages — 12.7 defines the visible part of a generic formal package. 6.a

The implicit empty private part is important because certain implicit declarations occur there if the package is a child package, and it defines types in its visible part that are derived from, or contain as components, private types declared within the parent package. These implicit declarations are visible in children of the child package. See 10.1.1. 6.b

7 [An entity declared in the private part of a package is visible only within the declarative region of the package itself (including any child units — see 10.1.1). In contrast, expanded names denoting entities declared in the visible part can be used even outside the package; furthermore, direct visibility of such entities can be achieved by means of use_clauses (see 4.1.3 and 8.4).]

Dynamic Semantics

8 The elaboration of a package_declaration consists of the elaboration of its basic_declarative_items in the given order.

9 NOTE 1 The visible part of a package contains all the information that another program unit is able to know about the package.

10 NOTE 2 If a declaration occurs immediately within the specification of a package, and the declaration has a corresponding completion that is a body, then that body has to occur immediately within the body of the package.

10.a **Proof:** This follows from the fact that the declaration and completion are required to occur immediately within the same declarative region, and the fact that bodies are disallowed (by the Syntax Rules) in package_specifications. This does not apply to instances of generic units, whose bodies can occur in package_specifications.

Examples

11 *Example of a package declaration:*

```
12    package Rational_Numbers is

13       type Rational is
            record
               Numerator   : Integer;
               Denominator : Positive;
            end record;

14       function "="(X,Y : Rational) return Boolean;

15       function "/"  (X,Y : Integer)  return Rational;   -- to construct a rational number

16       function "+"  (X,Y : Rational) return Rational;
         function "-"  (X,Y : Rational) return Rational;
         function "*"  (X,Y : Rational) return Rational;
         function "/"  (X,Y : Rational) return Rational;
    end Rational_Numbers;
```

17 There are also many examples of package declarations in the predefined language environment (see Annex A).

Incompatibilities With Ada 83

17.a/5 {*AI12-0417-1*} In Ada 83, a library package is allowed to have a body even if it doesn't need one. In Ada 95, a library package body is either required or forbidden — never optional. The workaround is to add aspect Elaborate_Body, or something else requiring a body, to each library package that has a body that isn't otherwise required.

Wording Changes from Ada 83

17.b/3 {*AI05-0299-1*} We have moved the syntax into this subclause and the next subclause from RM83-7.1, “Package Structure”, which we have removed.

17.c RM83 was unclear on the rules about when a package requires a body. For example, RM83-7.1(4) and RM83-7.1(8) clearly forgot about the case of an incomplete type declared in a package_declaration but completed in the body. In addition, RM83 forgot to make this rule apply to a generic package. We have corrected these rules. Finally, since we now allow a pragma Import for any explicit declaration, the completion rules need to take this into account as well.

Wording Changes from Ada 95

17.d/2 {*AI95-00420-01*} Defined “declaration list” to avoid ambiguity in other rules as to whether packages are included.

Extensions to Ada 2005

17.e/3 {*AI05-0183-1*} An optional aspect_specification can be used in a package_specification. This is described in 13.1.1.

7.2 Package Bodies

[In contrast to the entities declared in the visible part of a package, the entities declared in the package_body are visible only within the package_body itself. As a consequence, a package with a package_body can be used for the construction of a group of related subprograms in which the logical operations available to clients are clearly isolated from the internal entities.] 1

Syntax

{*AI05-0267-1*} package_body ::= 2/3
package body defining_program_unit_name
[aspect_specification] **is**
declarative_part
[**begin**
handled_sequence_of_statements]
end [[parent_unit_name.]identifier];

If an identifier or parent_unit_name.identifier appears at the end of a package_body, then this sequence of lexical elements shall repeat the defining_program_unit_name. 3

Legality Rules

{*AI12-0417-1*} A package_body shall be the completion of a previous package_declaration or generic_package_declaration. A library package_declaration or library generic_package_declaration shall not have a body unless it requires a body[; the Elaborate_Body aspect can be used to require a library_unit_declaration to have a body (see 10.2.1) if it would not otherwise require one]. 4/5

Ramification: The first part of the rule forbids a package_body from standing alone — it has to belong to some previous package_declaration or generic_package_declaration. 4.a

A nonlibrary package_declaration or nonlibrary generic_package_declaration that does not require a completion may have a corresponding body anyway. 4.b

Static Semantics

In any package_body without statements there is an implicit null_statement. For any package_declaration without an explicit completion, there is an implicit package_body containing a single null_statement. For a noninstance, nonlibrary package, this body occurs at the end of the declarative_part of the innermost enclosing program unit or block_statement; if there are several such packages, the order of the implicit package_bodies is unspecified. [(For an instance, the implicit package_body occurs at the place of the instantiation (see 12.3). For a library package, the place is partially determined by the elaboration dependences (see Clause 10).)] 5

Discussion: Thus, for example, we can refer to something happening just after the **begin** of a package_body, and we can refer to the handled_sequence_of_statements of a package_body, without worrying about all the optional pieces. The place of the implicit body makes a difference for tasks activated by the package. See also RM83-9.3(5). 5.a

The implicit body would be illegal if explicit in the case of a library package that does not require (and therefore does not allow) a body. This is a bit strange, but not harmful. 5.b

Dynamic Semantics

For the elaboration of a nongeneric package_body, its declarative_part is first elaborated, and its handled_sequence_of_statements is then executed. 6

NOTE 1 A variable declared in the body of a package is only visible within this body and, consequently, its value can only be changed within the package_body. In the absence of local tasks, the value of such a variable remains unchanged between calls issued from outside the package to subprograms declared in the visible part. The properties of such a variable are similar to those of a “static” variable of C. 7

8 NOTE 2 The elaboration of the body of a subprogram explicitly declared in the visible part of a package is caused by the elaboration of the body of the package. Hence a call of such a subprogram by an outside program unit raises the exception Program_Error if the call takes place before the elaboration of the package_body (see 3.11).

Examples

9 *Example of a package body (see 7.1):*

```
10    package body Rational_Numbers is

11       procedure Same_Denominator (X,Y : in out Rational) is
         begin
            -- reduces X and Y to the same denominator:
            ...
         end Same_Denominator;

12       function "="(X,Y : Rational) return Boolean is
            U : Rational := X;
            V : Rational := Y;
         begin
            Same_Denominator (U,V);
            return U.Numerator = V.Numerator;
         end "=";

13       function "/" (X,Y : Integer) return Rational is
         begin
            if Y > 0 then
               return (Numerator => X,  Denominator => Y);
            else
               return (Numerator => -X, Denominator => -Y);
            end if;
         end "/";

14       function "+" (X,Y : Rational) return Rational is ... end "+";
         function "-" (X,Y : Rational) return Rational is ... end "-";
         function "*" (X,Y : Rational) return Rational is ... end "*";
         function "/" (X,Y : Rational) return Rational is ... end "/";

15    end Rational_Numbers;
```

Wording Changes from Ada 83

15.a The syntax rule for package_body now uses the syntactic category handled_sequence_of_statements.

15.b The declarative_part of a package_body is now required; that doesn't make any real difference, since a declarative_part can be empty.

15.c RM83 seems to have forgotten to say that a package_body can't stand alone, without a previous declaration. We state that rule here.

15.d RM83 forgot to restrict the definition of elaboration of package_bodies to nongeneric ones. We have corrected that omission.

15.e The rule about implicit bodies (from RM83-9.3(5)) is moved here, since it is more generally applicable.

Extensions to Ada 2005

15.f/3 {*AI05-0267-1*} An optional aspect_specification can be used in a package_body. This is described in 13.1.1.

7.3 Private Types and Private Extensions

1 [The declaration (in the visible part of a package) of a type as a private type or private extension serves to separate the characteristics that can be used directly by outside program units (that is, the logical properties) from other characteristics whose direct use is confined to the package (the details of the definition of the type itself). See 3.9.1 for an overview of type extensions.]

Language Design Principles

1.a A private (untagged) type can be thought of as a record type with the type of its single (hidden) component being the full view.

A private tagged type can be thought of as a private extension of an anonymous parent with no components. The only dispatching operation of the parent is equality (although the Size attribute, and, if nonlimited, assignment are allowed, and those will presumably be implemented in terms of dispatching). 1.b

Syntax

{*AI05-0183-1*} private_type_declaration ::= 2/3
  type defining_identifier [discriminant_part] **is** [[**abstract**] **tagged**] [**limited**] **private**
    [aspect_specification];

{*AI95-00251-01*} {*AI95-00419-01*} {*AI95-00443-01*} {*AI05-0183-1*} 3/3
private_extension_declaration ::=
  type defining_identifier [discriminant_part] **is**
   [**abstract**] [**limited** | **synchronized**] **new** *ancestor_*subtype_indication
   [**and** interface_list] **with private**
    [aspect_specification];

Legality Rules

A private_type_declaration or private_extension_declaration declares a *partial view* of the type; such a declaration is allowed only as a declarative_item of the visible part of a package, and it requires a completion, which shall be a full_type_declaration that occurs as a declarative_item of the private part of the package. [The view of the type declared by the full_type_declaration is called the *full view*.] A generic formal private type or a generic formal private extension is also a partial view. 4

To be honest: A private type can also be imported (using aspect Import, see B.1), in which case no completion is allowed, if supported by an implementation. 4.a

Reason: We originally used the term "private view", but this was easily confused with the view provided *from* the private part, namely the full view. 4.b

Proof: {*AI95-00326-01*} Full view is now defined in 3.2.1, "Type Declarations", as all types now have them. 4.c/2

[A type shall be completely defined before it is frozen (see 3.11.1 and 13.14). Thus, neither the declaration of a variable of a partial view of a type, nor the creation by an allocator of an object of the partial view are allowed before the full declaration of the type. Similarly, before the full declaration, the name of the partial view cannot be used in a generic_instantiation or in a representation item.] 5

Proof: This rule is stated officially in 3.11.1, "Completions of Declarations". 5.a

{*AI95-00419-01*} {*AI95-00443-01*} [A private type is limited if its declaration includes the reserved word **limited**; a private extension is limited if its ancestor type is a limited type that is not an interface type, or if the reserved word **limited** or **synchronized** appears in its definition.] If the partial view is nonlimited, then the full view shall be nonlimited. If a tagged partial view is limited, then the full view shall be limited. [On the other hand, if an untagged partial view is limited, the full view may be limited or nonlimited.] 6/2

If the partial view is tagged, then the full view shall be tagged. [On the other hand, if the partial view is untagged, then the full view may be tagged or untagged.] In the case where the partial view is untagged and the full view is tagged, no derivatives of the partial view are allowed within the immediate scope of the partial view; [derivatives of the full view are allowed.] 7

Ramification: Note that deriving from a partial view within its immediate scope can only occur in a package that is a child of the one where the partial view is declared. The rule implies that in the visible part of a public child package, it is impossible to derive from an untagged private type declared in the visible part of the parent package in the case where the full view of the parent type turns out to be tagged. We considered a model in which the derived type was implicitly redeclared at the earliest place within its immediate scope where characteristics needed to be added. However, we rejected that model, because (1) it would imply that (for an untagged type) subprograms explicitly declared after the derived type could be inherited, and (2) to make this model work for composite types as well, several implicit redeclarations would be needed, since new characteristics can become visible one by one; that seemed like too much mechanism. 7.a

7.b **Discussion:** The rule for tagged partial views is redundant for partial views that are private extensions, since all extensions of a given ancestor tagged type are tagged, and limited if the ancestor is limited. We phrase this rule partially redundantly to keep its structure parallel with the other rules.

7.c **To be honest:** This rule is checked in a generic unit, rather than using the "assume the best" or "assume the worst" method.

7.d/2 **Reason:** {*AI95-00230-01*} Tagged limited private types have certain capabilities that are incompatible with having assignment for the full view of the type. In particular, tagged limited private types can be extended with components of a limited type, which works only because assignment is not allowed. Consider the following example:

7.e
```
package P1 is
    type T1 is tagged limited private;
    procedure Foo(X : in T1'Class);
private
    type T1 is tagged null record; -- Illegal!
        -- This should say "tagged limited null record".
end P1;
```

7.f/1
```
package body P1 is
    type A is access T1'Class;
    Global : A;
    procedure Foo(X : in T1'Class) is
    begin
        Global := new T1'Class'(X);
            -- This would be illegal if the full view of
            -- T1 were limited, like it's supposed to be.
    end Foo;
end P1;
```

7.g/2
```
{AI95-00230-01} with P1;
package P2 is
    type T2(D : access Integer)
            is new P1.T1 with
        record
            My_Task : Some_Task_Type; -- Trouble!
        end record;
end P2;
```

7.h/1
```
with P1;
with P2;
procedure Main is
    Local : aliased Integer;
    Y : P2.T2(D => Local'Access);
begin
    P1.Foo(Y);
end Main;
```

7.i/2 {*AI95-00230-01*} If the above example were legal, we would have succeeded in doing an assignment of a task object, which is supposed to be a no-no.

7.j This rule is not needed for private extensions, because they inherit their limitedness from their ancestor, and there is a separate rule forbidding limited components of the corresponding record extension if the parent is nonlimited.

7.k **Ramification:** A type derived from an untagged private type is untagged, even if the full view of the parent is tagged, and even at places that can see the parent:

7.l
```
package P is
    type Parent is private;
private
    type Parent is tagged
        record
            X: Integer;
        end record;
end P;
```

7.m/1
```
with P;
package Q is
    type T is new P.Parent;
end Q;
```

7.n
```
with Q; use Q;
package body P is
    ... T'Class ... -- Illegal!
    Object: T;
    ... Object.X ... -- Illegal!
    ... Parent(Object).X ... -- OK.
end P;
```

The declaration of T declares an untagged view. This view is always untagged, so T'Class is illegal, it would be illegal to extend T, and so forth. The component name X is never visible for this view, although the component is still there — one can get one's hands on it via a type_conversion. 7.o

{*AI95-00396-01*} If a full type has a partial view that is tagged, then: 7.1/2

- the partial view shall be a synchronized tagged type (see 3.9.4) if and only if the full type is a synchronized tagged type; 7.2/2

  Reason: Since we do not allow record extensions of synchronized tagged types, this property has to be visible in the partial view to avoid privacy breaking. Generic formals do not need a similar rule as any extensions are rechecked for legality in the specification, and extensions of tagged formals are always illegal in a generic body. 7.o.1/2

- the partial view shall be a descendant of an interface type (see 3.9.4) if and only if the full type is a descendant of the interface type. 7.3/2

  Reason: Consider the following example: 7.p/2

```
package P is                                                                    7.q/2
   package Pkg is
      type Ifc is interface;
      procedure Foo (X : Ifc) is abstract;
   end Pkg;

   type Parent_1 is tagged null record;                                         7.r/2

   type T1 is new Parent_1 with private;                                        7.s/2
private
   type Parent_2 is new Parent_1 and Pkg.Ifc with null record;
   procedure Foo (X : Parent_2); -- Foo #1

   type T1 is new Parent_2 with null record; -- Illegal.                        7.t/2
end P;

with P;                                                                         7.u/2
package P_Client is
   type T2 is new P.T1 and P.Pkg.Ifc with null record;
   procedure Foo (X : T2); -- Foo #2
   X : T2;
end P_Client;

with P_Client;                                                                  7.v/2
package body P is
   ...
{AI12-0005-1}    procedure Bar (X : T1'Class) is                              7.w/5
   begin
      Pkg.Foo (Pkg.Ifc'Class (X)); -- should call Foo #1 or an override thereof
   end;

begin                                                                           7.x/2
   Pkg.Foo (Pkg.Ifc'Class (P_Client.X));       -- should call Foo #2
   Bar (T1'Class (P_Client.X));
end P;
```

This example is illegal because the completion of T1 is descended from an interface that the partial view is not descended from. If it were legal, T2 would implement Ifc twice, once in the visible part of P, and once in the visible part of P_Client. We would need to decide how Foo #1 and Foo #2 relate to each other. There are two options: either Foo #2 overrides Foo #1, or it doesn't. 7.y/2

If Foo #2 overrides Foo #1, we have a problem because the client redefines a behavior that it doesn't know about, and we try to avoid this at all costs, as it would lead to a breakdown of whatever abstraction was implemented. If the abstraction didn't expose that it implements Ifc, there must be a reason, and it should be able to depend on the fact that no overriding takes place in clients. Also, during maintenance, things may change and the full view might implement a different set of interfaces. Furthermore, the situation is even worse if the full type implements another interface Ifc2 that happens to have a conforming Foo (otherwise unrelated, except for its name and profile). 7.z/2

If Foo #2 doesn't override Foo #1, there is some similarity with the case of normal tagged private types, where a client can declare an operation that happens to conform to some private operation, and that's OK, it gets a different slot in the type descriptor. The problem here is that T2 would implement Ifc in two different ways, and through conversions to Ifc'Class we could end up with visibility on both of these two different implementations. This is the “diamond inheritance” problem of C++ all over again, and we would need some kind of a preference rule to pick one implementation. We don't want to go there (if we did, we might as well provide full-fledged multiple inheritance). 7.aa/2

Note that there wouldn't be any difficulty to implement the first option, so the restriction is essentially methodological. The second option might be harder to implement, depending on the language rules that we would choose. 7.bb/2

7.cc/3 **Ramification:** {*AI05-0005-1*} This rule also prevents completing a private type with an interface. An interface, like all types, is a descendant of itself, and thus this rule is triggered. One reason this is necessary is that a client of a private extension should be able to inherit limitedness without having to look in the private part to see if the type is an interface (remember that limitedness of interfaces is never inherited, while it is inherited from other types).

8 The *ancestor subtype* of a private_extension_declaration is the subtype defined by the *ancestor_*-subtype_indication; the ancestor type shall be a specific tagged type. The full view of a private extension shall be derived (directly or indirectly) from the ancestor type. In addition to the places where Legality Rules normally apply (see 12.3), the requirement that the ancestor be specific applies also in the private part of an instance of a generic unit.

8.a **Reason:** This rule allows the full view to be defined through several intermediate derivations, possibly from a series of types produced by generic_instantiations.

8.1/2 {*AI95-00419-01*} {*AI95-00443-01*} If the reserved word **limited** appears in a private_extension_declaration, the ancestor type shall be a limited type. If the reserved word **synchronized** appears in a private_extension_declaration, the ancestor type shall be a limited interface.

9/5 {*AI12-0449-1*} If the declaration of a partial view includes a known_discriminant_part, then the full_type_declaration shall have a fully conforming [(explicit)] known_discriminant_part [(see 6.3.1)]. [The ancestor subtype may be unconstrained; the parent subtype of the full view is required to be constrained (see 3.7).]

9.a **Discussion:** If the ancestor subtype has discriminants, then it is usually best to make it unconstrained.

9.b **Ramification:** If the partial view has a known_discriminant_part, then the full view has to be a composite, non-array type, since only such types may have known discriminants. Also, the full view cannot inherit the discriminants in this case; the known_discriminant_part has to be explicit.

9.c That is, the following is illegal:

9.d
```
package P is
    type T(D : Integer) is private;
private
    type T is new Some_Other_Type; -- Illegal!
end P;
```

9.e even if Some_Other_Type has an integer discriminant called D.

9.f It is a ramification of this and other rules that in order for a tagged type to privately inherit unconstrained discriminants, the private type declaration has to have an unknown_discriminant_part.

10 If a private extension inherits known discriminants from the ancestor subtype, then the full view shall also inherit its discriminants from the ancestor subtype, and the parent subtype of the full view shall be constrained if and only if the ancestor subtype is constrained.

10.a **Reason:** The first part ensures that the full view has the same discriminants as the partial view. The second part ensures that if the partial view is unconstrained, then the full view is also unconstrained; otherwise, a client might constrain the partial view in a way that conflicts with the constraint on the full view.

10.1/3 {*AI95-00419-01*} {*AI05-0004-1*} If the full_type_declaration for a private extension includes a derived_type_definition, then the reserved word **limited** shall appear in the full_type_declaration if and only if it also appears in the private_extension_declaration.

10.b/3 **Reason:** {*AI05-0004-1*} The word **limited** is optional (unless the ancestor is an interface), but it should be used consistently. Otherwise things would be too confusing for the reader. Of course, we only require that if the full type includes a derived_type_definition, as we want to allow task and protected types to complete extensions of synchronized interfaces.

11 [If a partial view has unknown discriminants, then the full_type_declaration may define a definite or an indefinite subtype, with or without discriminants.]

12 If a partial view has neither known nor unknown discriminants, then the full_type_declaration shall define a definite subtype.

If the ancestor subtype of a private extension has constrained discriminants, then the parent subtype of the full view shall impose a statically matching constraint on those discriminants. 13

Ramification: If the parent type of the full view is not the ancestor type, but is rather some descendant thereof, the constraint on the discriminants of the parent type might come from the declaration of some intermediate type in the derivation chain between the ancestor type and the parent type. 13.a

Reason: This prevents the following: 13.b

```
package P is                                                                    13.c
    type T2 is new T1(Discrim => 3) with private;
private
    type T2 is new T1(Discrim => 999) -- Illegal!
        with record ...;
end P;
```

The constraints in this example do not statically match. 13.d

If the constraint on the parent subtype of the full view depends on discriminants of the full view, then the ancestor subtype has to be unconstrained: 13.e

```
type One_Discrim(A: Integer) is tagged ...;                                     13.f
...
package P is
    type Two_Discrims(B: Boolean; C: Integer) is new One_Discrim with private;
private
    type Two_Discrims(B: Boolean; C: Integer) is new One_Discrim(A => C) with
        record
            ...
        end record;
end P;
```

The above example would be illegal if the private extension said "is new One_Discrim(A => C);", because then the constraints would not statically match. (Constraints that depend on discriminants are not static.) 13.g

Static Semantics

A private_type_declaration declares a private type and its first subtype. Similarly, a private_extension_-declaration declares a private extension and its first subtype. 14

Discussion: A *package-private type* is one declared by a private_type_declaration; that is, a private type other than a generic formal private type. Similarly, a *package-private extension* is one declared by a private_extension_declaration. These terms are not used in the RM95 version of this document. 14.a

{*AI05-0269-1*} A declaration of a partial view and the corresponding full_type_declaration define two views of a single type. The declaration of a partial view together with the visible part define the operations that are available to outside program units; the declaration of the full view together with the private part define other operations whose direct use is possible only within the declarative region of the package itself. Moreover, within the scope of the declaration of the full view, the characteristics (see 3.4) of the type are determined by the full view; in particular, within its scope, the full view determines the classes that include the type, which components, entries, and protected subprograms are visible, what attributes and other predefined operations are allowed, and whether the first subtype is static. See 7.3.1. 15/3

{*AI95-00401-01*} {*AI05-0110-1*} For a private extension, the characteristics (including components, but excluding discriminants if there is a new discriminant_part specified), predefined operators, and inherited user-defined primitive subprograms are determined by its ancestor type and its progenitor types (if any), in the same way that those of a record extension are determined by those of its parent type and its progenitor types (see 3.4 and 7.3.1). 16/3

To be honest: {*AI05-0110-1*} If an operation of the ancestor or parent type is abstract, then the abstractness of the inherited operation is different for nonabstract record extensions than for nonabstract private extensions (see 3.9.3). 16.a/3

Dynamic Semantics

The elaboration of a private_type_declaration creates a partial view of a type. The elaboration of a private_extension_declaration elaborates the *ancestor*_subtype_indication, and creates a partial view of a type. 17

18/5 NOTE 1 {*AI12-0442-1*} The partial view of a type as declared by a private_type_declaration is defined to be a composite view (in 3.2). The full view of the type can be elementary or composite. A private extension is also composite, as is its full view.

19/2 NOTE 2 {*AI95-00318-02*} Declaring a private type with an unknown_discriminant_part is a way of preventing clients from creating uninitialized objects of the type; they are then forced to initialize each object by calling some operation declared in the visible part of the package.

19.a **Discussion:** Packages with private types are analogous to generic packages with formal private types, as follows: The declaration of a package-private type is like the declaration of a formal private type. The visible part of the package is like the generic formal part; these both specify a contract (that is, a set of operations and other things available for the private type). The private part of the package is like an instantiation of the generic; they both give a full_type_declaration that specifies implementation details of the private type. The clients of the package are like the body of the generic; usage of the private type in these places is restricted to the operations defined by the contract.

19.b In other words, being inside the package is like being outside the generic, and being outside the package is like being inside the generic; a generic is like an "inside-out" package.

19.c This analogy also works for private extensions in the same inside-out way.

19.d Many of the legality rules are defined with this analogy in mind. See, for example, the rules relating to operations of [formal] derived types.

19.e The completion rules for a private type are intentionally quite similar to the matching rules for a generic formal private type.

19.f This analogy breaks down in one respect: a generic actual subtype is a subtype, whereas the full view for a private type is always a new type. (We considered allowing the completion of a private_type_declaration to be a subtype_declaration, but the semantics just won't work.) This difference is behind the fact that a generic actual type can be class-wide, whereas the completion of a private type always declares a specific type.

20/5 NOTE 3 {*AI95-00401*} {*AI12-0442-1*} The ancestor type specified in a private_extension_declaration and the parent type specified in the corresponding declaration of a record extension given in the private part can be different. If the ancestor type is not an interface type, the parent type of the full view can be any descendant of the ancestor type. In this case, for a primitive subprogram that is inherited from the ancestor type and not overridden, the formal parameter names and default expressions (if any) come from the corresponding primitive subprogram of the specified ancestor type, while the body comes from the corresponding primitive subprogram of the parent type of the full view. See 3.9.2.

21/5 NOTE 4 {*AI95-00401*} {*AI12-0442-1*} If the ancestor type specified in a private_extension_declaration is an interface type, the parent type can be any type so long as the full view is a descendant of the ancestor type. The progenitor types specified in a private_extension_declaration and the progenitor types specified in the corresponding declaration of a record extension given in the private part are not necessarily the same — it is only necessary that the private extension and the record extension be descended from the same set of interfaces.

Examples

22 *Examples of private type declarations:*

23
```
type Key is private;
type File_Name is limited private;
```

24 *Example of a private extension declaration:*

25
```
type List is new Ada.Finalization.Controlled with private;
```

Extensions to Ada 83

25.a The syntax for a private_type_declaration is augmented to allow the reserved word **tagged**.

25.b In Ada 83, a private type without discriminants cannot be completed with a type with discriminants. Ada 95 allows the full view to have discriminants, so long as they have defaults (that is, so long as the first subtype is definite). This change is made for uniformity with generics, and because the rule as stated is simpler and easier to remember than the Ada 83 rule. In the original version of Ada 83, the same restriction applied to generic formal private types. However, the restriction was removed by the ARG for generics. In order to maintain the "generic contract/private type contract analogy" discussed above, we have to apply the same rule to package-private types. Note that a private untagged type without discriminants can be completed with a tagged type with discriminants only if the full view is constrained, because discriminants of tagged types cannot have defaults.

Wording Changes from Ada 83

25.c RM83-7.4.1(4), "Within the specification of the package that declares a private type and before the end of the corresponding full type declaration, a restriction applies....", is subsumed (and corrected) by the rule that a type shall

be completely defined before it is frozen, and the rule that the parent type of a derived type declaration shall be completely defined, unless the derived type is a private extension.

Extensions to Ada 95

{*AI95-00251-01*} {*AI95-00396-01*} {*AI95-00401-01*} Added interface_list to private extensions to support interfaces and multiple inheritance (see 3.9.4). 25.d/2

{*AI95-00419-01*} A private extension may specify that it is a limited type. This is required for interface ancestors (from which limitedness is not inherited), but it is generally useful as documentation of limitedness. 25.e/2

{*AI95-00443-01*} A private extension may specify that it is a synchronized type. This is required in order so that a regular limited interface can be used as the ancestor of a synchronized type (we do not allow hiding of synchronization). 25.f/2

Extensions to Ada 2005

{*AI05-0183-1*} An optional aspect_specification can be used in a private_type_declaration and a private_extension_declaration. This is described in 13.1.1. 25.g/3

Wording Changes from Ada 2005

{*AI05-0110-1*} **Correction:** The description of how a private extension inherits characteristics was made consistent with the way formal derived types inherit characteristics (see 12.5.1). 25.h/3

7.3.1 Private Operations

[For a type declared in the visible part of a package or generic package, certain operations on the type do not become visible until later in the package — either in the private part or the body. Such *private operations* are available only inside the declarative region of the package or generic package.] 1

Static Semantics

The predefined operators that exist for a given type are determined by the classes to which the type belongs. For example, an integer type has a predefined "+" operator. In most cases, the predefined operators of a type are declared immediately after the definition of the type; the exceptions are explained below. Inherited subprograms are also implicitly declared immediately after the definition of the type, except as stated below. 2

{*8652/0019*} {*AI95-00033-01*} {*AI05-0029-1*} For a composite type, the characteristics (see 7.3) of the type are determined in part by the characteristics of its component types. At the place where the composite type is declared, the only characteristics of component types used are those characteristics visible at that place. If later immediately within the declarative region in which the composite type is declared additional characteristics become visible for a component type, then any corresponding characteristics become visible for the composite type. Any additional predefined operators are implicitly declared at that place. If there is no such place, then additional predefined operators are not declared at all, but they still exist. 3/3

Reason: {*AI05-0029-1*} We say that the predefined operators exist because they can emerge in some unusual generic instantiations. See 12.5. 3.a/3

Discussion: {*AI05-0029-1*} The predefined operators for the underlying class of a type always exist, even if there is no visibility on that underlying class. This rule is simply about where (if ever) those operators are declared (and thus become usable). The “additional predefined operators” defined by this rule are any that are not declared at the point of the original type declaration. For instance, a type derived from a private type whose full type is type String always will have a ">" operator, but where that operator is declared (and thus whether it is visible) will depend on the visibility of the full type of the parent type. 3.b/3

{*8652/0019*} {*AI95-00033-01*} The corresponding rule applies to a type defined by a derived_type_definition, if there is a place immediately within the declarative region in which the type is declared where additional characteristics of its parent type become visible. 4/1

5/1 {*8652/0019*} {*AI95-00033-01*} [For example, an array type whose component type is limited private becomes nonlimited if the full view of the component type is nonlimited and visible at some later place immediately within the declarative region in which the array type is declared. In such a case, the predefined "=" operator is implicitly declared at that place, and assignment is allowed after that place.]

5.1/5 {*AI12-0140-1*} The characteristics and constraints of the designated subtype of an access type follow a somewhat different rule. The view of the designated subtype of (a view of) an access type at a given place is determined by the view of the designated subtype that is visible at that place, rather than the view at the place where the access type is declared.

5.a.1/5 **Ramification:** {*AI12-0140-1*} Whether or not the designated subtype is considered incomplete is determined by rules in 3.10.1; this rule has no effect on that property.

5.2/5 {*AI05-0115-1*} {*AI05-0269-1*} {*AI12-0140-1*} A type is a *descendant* of the full view of some ancestor of its parent type only if the current view it has of its parent is a descendant of the full view of that ancestor. More generally, at any given place, a type is descended from the same view of an ancestor as that from which the current view of its parent is descended. This view determines what characteristics are inherited from the ancestor[, and, for example, whether the type is considered to be a descendant of a record type, or a descendant only through record extensions of a more distant ancestor].

5.3/5 {*AI05-0115-1*} {*AI12-0065-1*} {*AI12-0140-1*} {*AI12-0451-1*} [Furthermore, it is possible for there to be places where a derived type is known to be derived indirectly from an ancestor type, but is not a descendant of even a partial view of the ancestor type, because the parent of this derived type is not visibly a descendant of the ancestor. In this case, the derived type inherits no characteristics from that ancestor, but nevertheless is within the derivation class of the ancestor for the purposes of type conversion, the "covers" relationship, and matching against a formal derived type. In this case the derived type is effectively a *descendant* of an incomplete view of the ancestor.]

5.a/3 **Discussion:** Here is an example of this situation:

5.b/3
```
package P is
   type T is private;
   C : constant T;
private
   type T is new Integer;
   C : constant T := 42;
end P;
```

5.c/4
```
{AI12-0065-1} with P;
package Q is
    type T2 is new P.T;   -- T2 is not a descendant of Integer
end Q;
```

5.d/4
```
{AI12-0065-1} with Q;
package P.Child is
    type T3 is new Q.T2;
private
    -- Here T3 is known to be indirectly derived from Integer, but inherits
    -- no characteristics from Integer, since T2 inherits no characteristics
    -- from Integer.
    -- However, we allow an explicit conversion of T3 to/from Integer.
    -- Hence, T3 is effectively a descendant of an "incomplete" view of Integer.
    Int : Integer := 52;
    V : T3 := T3(P.C);   -- Legal: conversion allowed
    W : T3 := T3(Int);   -- Legal: conversion allowed
    X : T3 := T3(42);    -- Error: T3 is not a numeric type
    Y : T3 := X + 1;     -- Error: no visible "+" operator
    Z : T3 := T3(Integer(W) + 1);    -- Legal: convert to Integer first
end P.Child;
```

6/3 {*8652/0019*} {*AI95-00033-01*} {*AI05-0029-1*} Inherited primitive subprograms follow a different rule. For a derived_type_definition, each inherited primitive subprogram is implicitly declared at the earliest place, if any, immediately within the declarative region in which the type_declaration occurs, but after the type_declaration, where the corresponding declaration from the parent is visible. If there is no such

place, then the inherited subprogram is not declared at all, but it still exists. [For a tagged type, it is possible to dispatch to an inherited subprogram that is not declared at all.]

{*AI12-0439-1*} For a private_extension_declaration, each inherited subprogram is declared immediately after the private_extension_declaration if the corresponding declaration from the ancestor is visible at that place. Otherwise, the inherited subprogram is not declared for the private extension, [though it can be for the full type]. 7/5

Reason: There is no need for the "earliest place immediately within the declarative region" business here, because a private_extension_declaration will be completed with a full_type_declaration, so we can hang the necessary private implicit declarations on the full_type_declaration. 7.a/1

Discussion: The above rules matter only when the component type (or parent type) is declared in the visible part of a package, and the composite type (or derived type) is declared within the declarative region of that package (possibly in a nested package or a child package). 7.b

Consider: 7.c

```
package Parent is                                                    7.d
    type Root is tagged null record;
    procedure Op1(X : Root);

    type My_Int is range 1..10;                                      7.e
private
    procedure Op2(X : Root);

    type Another_Int is new My_Int;                                  7.f
    procedure Int_Op(X : My_Int);
end Parent;

with Parent; use Parent;                                             7.g
package Unrelated is
    type T2 is new Root with null record;
    procedure Op2(X : T2);
end Unrelated;

package Parent.Child is                                              7.h
    type T3 is new Root with null record;
    -- Op1(T3) implicitly declared here.

    package Nested is                                                7.i
        type T4 is new Root with null record;
    private
        ...
    end Nested;
private
    -- Op2(T3) implicitly declared here.
    ...
end Parent.Child;

with Unrelated; use Unrelated;                                       7.j
package body Parent.Child is
    package body Nested is
        -- Op2(T4) implicitly declared here.
    end Nested;

    type T5 is new T2 with null record;                              7.k
end Parent.Child;
```

Another_Int does not inherit Int_Op, because Int_Op does not "exist" at the place where Another_Int is declared. 7.l

Type T2 inherits Op1 and Op2 from Root. However, the inherited Op2 is never declared, because Parent.Op2 is never visible immediately within the declarative region of T2. T2 explicitly declares its own Op2, but this is unrelated to the inherited one — it does not override the inherited one, and occupies a different slot in the type descriptor. 7.m/1

T3 inherits both Op1 and Op2. Op1 is implicitly declared immediately after the type declaration, whereas Op2 is declared at the beginning of the private part. Note that if Child were a private child of Parent, then Op1 and Op2 would both be implicitly declared immediately after the type declaration. 7.n

T4 is similar to T3, except that the earliest place immediately within the declarative region containing T4 where Root's Op2 is visible is in the body of Nested. 7.o/1

If T3 or T4 were to declare a type-conformant Op2, this would override the one inherited from Root. This is different from the situation with T2. 7.p

7.q T5 inherits Op1 and two Op2's from T2. Op1 is implicitly declared immediately after the declaration of T5, as is the Op2 that came from Unrelated.Op2. However, the Op2 that originally came from Parent.Op2 is never implicitly declared for T5, since T2's version of that Op2 is never visible (anywhere — it never got declared either).

7.r For all of these rules, implicit private parts and bodies are assumed as needed.

7.s It is possible for characteristics of a type to be revealed in more than one place:

7.t
```
package P is
    type Comp1 is private;
private
    type Comp1 is new Boolean;
end P;
```

7.u
```
package P.Q is
    package R is
        type Comp2 is limited private;
        type A is array(Integer range <>) of Comp2;
    private
        type Comp2 is new Comp1;
        -- A becomes nonlimited here.
        -- "="(A, A) return Boolean is implicitly declared here.
        ...
    end R;
private
    -- Now we find out what Comp1 really is, which reveals
    -- more information about Comp2, but we're not within
    -- the immediate scope of Comp2, so we don't do anything
    -- about it yet.
end P.Q;
```

7.v
```
package body P.Q is
    package body R is
        -- Things like "xor"(A,A) return A are implicitly
        -- declared here.
    end R;
end P.Q;
```

7.v.1/1 {*8652/0019*} {*AI95-00033-01*} We say *immediately* within the declarative region in order that types do not gain operations within a nested scope. Consider:

7.v.2/1
```
package Outer is
    package Inner is
        type Inner_Type is private;
    private
        type Inner_Type is new Boolean;
    end Inner;
    type Outer_Type is array(Natural range <>) of Inner.Inner_Type;
end Outer;
```

7.v.3/1
```
package body Outer is
    package body Inner is
        -- At this point, we can see that Inner_Type is a Boolean type.
        -- But we don't want Outer_Type to gain an "and" operator here.
    end Inner;
end Outer;
```

8 [The Class attribute is defined for tagged subtypes in 3.9. In addition,] for every subtype S of an untagged private type whose full view is tagged, the following attribute is defined:

9 S'Class Denotes the class-wide subtype corresponding to the full view of S. This attribute is allowed only from the beginning of the private part in which the full view is declared, until the declaration of the full view. [After the full view, the Class attribute of the full view can be used.]

10 NOTE 1 Because a partial view and a full view are two different views of one and the same type, outside of the defining package the characteristics of the type are those defined by the visible part. Within these outside program units the type is just a private type or private extension, and any language rule that applies only to another class of types does not apply. The fact that the full declaration can implement a private type with a type of a particular class (for example, as an array type) is relevant only within the declarative region of the package itself including any child units.

11 The consequences of this actual implementation are, however, valid everywhere. For example: any default initialization of components takes place; the attribute Size provides the size of the full view; finalization is still done for controlled components of the full view; task dependence rules still apply to components that are task objects.

NOTE 2 {*AI95-00287-01*} Partial views provide initialization, membership tests, selected components for the selection of discriminants and inherited components, qualification, and explicit conversion. Nonlimited partial views also allow use of assignment_statements. 12/2

NOTE 3 For a subtype S of a partial view, S'Size is defined (see 13.3). For an object A of a partial view, the attributes A'Size and A'Address are defined (see 13.3). The Position, First_Bit, and Last_Bit attributes are also defined for discriminants and inherited components. 13

Examples

Example of a type with private operations: 14

```
package Key_Manager is                                                          15
   type Key is private;
   Null_Key : constant Key; -- a deferred constant declaration (see 7.4)
   procedure Get_Key(K : out Key);
   function "<" (X, Y : Key) return Boolean;
private
   type Key is new Natural;
   Null_Key : constant Key := Key'First;
end Key_Manager;

package body Key_Manager is                                                     16
   Last_Key : Key := Null_Key;
   procedure Get_Key(K : out Key) is
   begin
      Last_Key := Last_Key + 1;
      K := Last_Key;
   end Get_Key;

   function "<" (X, Y : Key) return Boolean is                                  17
   begin
      return Natural(X) < Natural(Y);
   end "<";
end Key_Manager;
```

{*AI12-0452-1*} Outside of the package Key_Manager, the operations available for objects of type Key include assignment, the comparison for equality or inequality, the procedure Get_Key and the operator "<"; they do not include other relational operators such as ">=", or arithmetic operators. 18/5

{*AI12-0440-1*} {*AI12-0452-1*} The explicitly declared operator "<" hides the predefined operator "<" implicitly declared by the full_type_declaration. Within the body of the function, an explicit conversion of X and Y to the subtype Natural is necessary to invoke the "<" operator of the parent type. Alternatively, the result of the function can be written as **not** (X >= Y), since the operator ">=" is not redefined. 19/5

{*AI12-0452-1*} The value of the variable Last_Key, declared in the package body, remains unchanged between calls of the procedure Get_Key. (See also the NOTEs of 7.2.) 20/5

Wording Changes from Ada 83

The phrase in RM83-7.4.2(7), "...after the full type declaration", doesn't work in the presence of child units, so we define that rule in terms of visibility. 23.a

The definition of the Constrained attribute for private types has been moved to "Obsolescent Features". (The Constrained attribute of an object has not been moved there.) 23.b

Wording Changes from Ada 95

{*8652/0018*} {*AI95-00033-01*} **Corrigendum:** Clarified when additional operations are declared. 23.c/2

{*AI95-00287-01*} Revised the note on operations of partial views to reflect that limited types do have an assignment operation, but not assignment_statements. 23.d/2

Wording Changes from Ada 2005

{*AI05-0029-1*} **Correction:** Revised the wording to say that predefined operations still exist even if they are never declared, because it is possible to reference them in a generic unit. 23.e/3

23.f/3 {*AI05-0115-1*} **Correction:** Clarified that the characteristics of a descendant of a private type depend on the visibility of the full view of the direct ancestor. This has to be the case (so that privacy is not violated), but it wasn't spelled out in earlier versions of Ada.

Wording Changes from Ada 2012

23.g/4 {*AI12-0065-1*} **Corrigendum:** Clarified the clarification added by AI05-0115-1, as it turned out to not be that clear. Hopefully this version is better.

23.h/5 {*AI12-0140-1*} **Correction:** Clarified the constraints and properties that apply to a designated subtype. This additional wording does not mean to change existing practice.

7.3.2 Type Invariants

1/5 {*AI05-0146-1*} {*AI12-0041-1*} {*AI12-0396-1*} For a private type, private extension, or interface, the following language-defined assertion aspects may be specified with an aspect_specification (see 13.1.1):

2/3 {*AI05-0146-1*} {*AI05-0250-1*} Type_Invariant

This aspect shall be specified by an expression, called an *invariant expression*. Type_Invariant may be specified on a private_type_declaration, on a private_extension_declaration, or on a full_type_declaration that declares the completion of a private type or private extension.

2.a/3 **Aspect Description for Type_Invariant:** A condition that will hold true for all objects of a type.

3/5 {*AI05-0146-1*} {*AI12-0041-1*} {*AI12-0150-1*} {*AI12-0419-1*} Type_Invariant'Class

This aspect shall be specified by an expression, called an *invariant expression*. Type_Invariant'Class may be specified on a private_type_declaration, a private_extension_declaration, or a full_type_declaration for an interface type. Type_Invariant'Class determines a *class-wide type invariant* for a tagged type. [The Type_Invariant'Class aspect is not inherited, but its effects are additive, as defined below.]

3.a/3 **Reason:** {*AI05-0254-1*} A class-wide type invariant cannot be hidden in the private part, as the creator of an extension needs to know about it in order to conform to it in any new or overriding operations. On the other hand, a specific type invariant is not inherited, so that no operation outside of the original package needs to conform to it; thus there is no need for it to be visible.

3.b/3 **Aspect Description for Type_Invariant'Class:** A condition that will hold true for all objects in a class of types.

3.c/5 **Term entry: invariant** — assertion that is expected to be True for all objects of a given private type when viewed from outside the defining package

Name Resolution Rules

4/3 {*AI05-0146-1*} The expected type for an invariant expression is any boolean type.

5/5 {*AI05-0146-1*} {*AI12-0150-1*} {*AI12-0159-1*} {*AI12-0199-1*} [Within an invariant expression, the identifier of the first subtype of the associated type denotes the current instance of the type.] Within an invariant expression for the Type_Invariant aspect of a type *T*, the type of this current instance is *T*. Within an invariant expression for the Type_Invariant'Class aspect of a type *T*, the type of this current instance is interpreted as though it had a (notional) nonabstract type *NT* that is a visible formal derived type whose ancestor type is *T*.[The effect of this interpretation is that the only operations that can be applied to this current instance are those defined for such a formal derived type.]

5.a/3 **Proof:** The first sentence is given formally in 13.1.1.

5.b/4 **Reason:** {*AI12-0159-1*} The rules for Type_Invariant'Class ensure that the invariant expression is well-defined for any type descended from *T*.

Legality Rules

6/3 {*AI05-0146-1*} [The Type_Invariant'Class aspect shall not be specified for an untagged type.] The Type_Invariant aspect shall not be specified for an abstract type.

Proof: The first sentence is given formally in 13.1.1. 6.a/3

{*AI12-0042-1*} {*AI12-0382-1*} If a type extension occurs immediately within the visible part of a package specification, at a point where a private operation of some ancestor is visible and inherited, and a Type_Invariant'Class expression applies to that ancestor, then the inherited operation shall be abstract or shall be overridden. 6.1/5

Static Semantics

{*AI05-0250-1*} [If the Type_Invariant aspect is specified for a type *T*, then the invariant expression applies to *T*.] 7/3

{*AI05-0146-1*} {*AI12-0199-1*} If the Type_Invariant'Class aspect is specified for a tagged type *T*, then a *corresponding expression* also applies to each nonabstract descendant *T1* of *T* [(including *T* itself if it is nonabstract)]. The corresponding expression is constructed from the associated expression as follows: 8/5

- {*AI12-0199-1*} References to nondiscriminant components of *T* (or to *T* itself) are replaced with references to the corresponding components of *T1* (or to *T1* as a whole). 8.1/5

  Ramification: {*AI12-0199-1*} The only nondiscriminant components visible at the point of such an aspect specification are necessarily inherited from some nonprivate ancestor. 8.a.1/5

- {*AI12-0199-1*} References to discriminants of *T* are replaced with references to the corresponding discriminant of *T1*, or to the specified value for the discriminant, if the discriminant is specified by the derived_type_definition for some type that is an ancestor of *T1* and a descendant of *T* (see 3.7). 8.2/5

  This paragraph was deleted. 8.a/5

  Discussion: The associated expression from which the corresponding expression is constructed is the one that applies to the descendant type; "applies" is formally defined in 13.1.1. 8.b/5

{*AI12-0075-1*} {*AI12-0191-1*} For a nonabstract type *T*, a callable entity is said to be a *boundary entity* for *T* if it is declared within the immediate scope of *T* (or by an instance of a generic unit, and the generic is declared within the immediate scope of type *T*), or is the Read or Input stream-oriented attribute of type *T*, and either: 8.3/5

- *T* is a private type or a private extension and the callable entity is visible outside the immediate scope of type T or overrides an inherited operation that is visible outside the immediate scope of *T*; or 8.4/5

- *T* is a record extension, and the callable entity is a primitive operation visible outside the immediate scope of type *T* or overrides an inherited operation that is visible outside the immediate scope of *T*. 8.5/5

  Reason: {*AI12-0191-1*} A boundary entity for type *T* is one that might require an invariant check for *T*. It includes subprograms that don't (visibly) involve T; since we don't want to break privacy, we can't statically know if some private type has some part of *T*. We'll reduce the set when we describe the actual checks. 8.c/5

Dynamic Semantics

{*AI05-0146-1*} {*AI05-0247-1*} {*AI05-0290-1*} {*AI12-0150-1*} If one or more invariant expressions apply to a nonabstract type *T*, then an invariant check is performed at the following places, on the specified object(s): 9/4

- {*AI12-0133-1*} After successful initialization of an object of type *T* by default (see 3.3.1), the check is performed on the new object unless the partial view of *T* has unknown discriminants; 10/4

  Reason: {*AI12-0133-1*} The check applies everywhere, even in the package body, because default initialization has to work the same for clients as it does within the package. As such, checks within the package are either harmless or will uncover a bug that could also happen to a client. However, if the partial view of the type has unknown discriminants, no client of the package can declare a default-initialized object. Therefore, no invariant check is needed, as all default initialized objects are necessarily inside the package. 10.a/4

10.1/5 • {*AI12-0049-1*} {*AI12-0191-1*} After successful explicit initialization of the completion of a deferred constant whose nominal type has a part of type *T*, if the completion is inside the immediate scope of the full view of *T*, and the deferred constant is visible outside the immediate scope of *T*, the check is performed on the part(s) of type *T*;

11/3 • After successful conversion to type *T*, the check is performed on the result of the conversion;

12/3 • {*AI05-0146-1*} {*AI05-0269-1*} For a view conversion, outside the immediate scope of *T*, that converts from a descendant of *T* (including *T* itself) to an ancestor of type *T* (other than *T* itself), a check is performed on the part of the object that is of type *T*:

13/3 - after assigning to the view conversion; and

14/3 - after successful return from a call that passes the view conversion as an **in out** or **out** parameter.

14.a/3 **Ramification:** For a single view conversion that converts between distantly related types, this rule could be triggered for multiple types and thus multiple invariant checks may be needed.

14.b/3 **Implementation Note:** {*AI05-0299-1*} For calls to inherited subprograms (including dispatching calls), the implied view conversions mean that a wrapper is probably needed. (See the Note at the bottom of this subclause for more on the model of checks for inherited subprograms.)

14.c/3 For view conversions involving class-wide types, the exact checks needed may not be known at compile-time. One way to deal with this is to have an implicit dispatching operation that is given the object to check and the tag of the target of the conversion, and which first checks if the passed tag is not for itself, and if not, checks the its invariant on the object and then calls the operation of its parent type. If the tag is for itself, the operation is complete.

15/5 • {*AI12-0146-1*} {*AI12-0075-1*} {*AI12-0191-1*} {*AI12-0193-1*} Upon successful return from a call on any callable entity which is a boundary entity for *T*, an invariant check is performed on each object which is subject to an invariant check for *T*. In the case of a call to a protected operation, the check is performed before the end of the protected action. In the case of a call to a task entry, the check is performed before the end of the rendezvous. The following objects of a callable entity are subject to an invariant check for *T*:

Paragraph 16 was merged above.

17/5 - {*AI05-0146-1*} {*AI05-0269-1*} {*AI12-0042-1*} {*AI12-0075-1*} {*AI12-0191-1*} a result with a nominal type that has a part of type *T*;

18/5 - {*AI12-0042-1*} {*AI12-0191-1*} an **out** or **in out** parameter whose nominal type has a part of type *T*;

18.1/5 - {*AI12-0075-1*} {*AI12-0191-1*} an access-to-object parameter or result whose designated nominal type has a part of type *T*; or

19/5 - {*AI05-0289-1*} {*AI12-0042-1*} {*AI12-0044-1*} {*AI12-0075-1*} {*AI12-0191-1*} for a procedure or entry, an **in** parameter whose nominal type has a part of type *T*.

19.a/5 **Ramification:** {*AI12-0167-1*} {*AI12-0191-1*} This is a Dynamic Semantics rule, so we ignore privacy when determining if a check is needed. We do, however, use the nominal type of objects to determine if a part of type *T* is present; therefore, parts that aren't known at compile-time (after ignoring privacy) are never subject to an invariant check. This is preferable, as we don't want overhead associated with the possibility that there *might* exist an extension of a tagged type that has a part of type *T*. (See the "leaks" note below for avoidance advice.)

19.b/4 **Discussion:** We don't check **in** parameters for functions to avoid infinite recursion for calls to public functions appearing in invariant expressions. Such function calls are unavoidable for class-wide invariants and likely for other invariants. This is the simplest rule that avoids trouble, and functions are much more likely to be queries that don't modify their parameters than other callable entities.

20/5 {*AI05-0146-1*} {*AI05-0269-1*} {*AI12-0075-1*} {*AI12-0338-1*} If the nominal type of a formal parameter (or the designated nominal type of an access-to-object parameter or result) is incomplete at the point of the declaration of the callable entity, and if the completion of that incomplete type does not occur in the same declaration list as the incomplete declaration, then for purposes of the preceding rules the nominal type is considered to have no parts of type *T*.

- {*AI12-0042-1*} For a view conversion to a class-wide type occurring within the immediate scope of *T*, from a specific type that is a descendant of *T* (including *T* itself), a check is performed on the part of the object that is of type *T*. 20.1/4

  Reason: Class-wide objects are treated as though they exist outside the scope of every type, and may be passed across package "boundaries" freely without further invariant checks. 20.a/4

  {*AI12-0167-1*} Despite this model, if an object of type *T* that is a component of a class-wide object is modified within the scope of the full view of type *T*, then there is no invariant check for *T* at that point. 20.b/5

{*AI05-0290-1*} {*AI12-0080-1*} {*AI12-0159-1*} If performing checks is required by the Type_Invariant or Type_Invariant'Class assertion policies (see 11.4.2) in effect at the point of the corresponding aspect specification applicable to a given type, then the respective invariant expression is considered *enabled*. 21/4

Ramification: If a class-wide invariant expression is enabled for a type, it remains enabled when inherited by descendants of that type, even if the policy in effect is Ignore for the inheriting type. 21.a/3

{*AI05-0146-1*} {*AI05-0250-1*} {*AI05-0289-1*} {*AI05-0290-1*} The invariant check consists of the evaluation of each enabled invariant expression that applies to *T*, on each of the objects specified above. If any of these evaluate to False, Assertions.Assertion_Error is raised at the point of the object initialization, conversion, or call. If a given call requires more than one evaluation of an invariant expression, either for multiple objects of a single type or for multiple types with invariants, the evaluations are performed in an arbitrary order, and if one of them evaluates to False, it is not specified whether the others are evaluated. Any invariant check is performed prior to copying back any by-copy **in out** or **out** parameters. Invariant checks, any postcondition check, and any constraint or predicate checks associated with **in out** or **out** parameters are performed in an arbitrary order. 22/3

{*AI12-0150-1*} {*AI12-0159-1*} For an invariant check on a value of type *T1* based on a class-wide invariant expression inherited from an ancestor type *T*, any operations within the invariant expression that were resolved as primitive operations of the (notional) formal derived type *NT* are bound to the corresponding operations of type *T1* in the evaluation of the invariant expression for the check on *T1*. 22.1/4

{*AI05-0146-1*} {*AI05-0247-1*} {*AI05-0250-1*} The invariant checks performed on a call are determined by the subprogram or entry actually invoked, whether directly, as part of a dispatching call, or as part of a call through an access-to-subprogram value. 23/3

Ramification: {*AI12-0149-1*} {*AI12-0167-1*} {*AI12-0210-1*} Type invariant checks are intended to prevent invariant-violating values from inadvertently "leaking out"; that is, code which cannot see the completion of the private type should not be able to reference invariant-violating values of the type (assuming the type invariant condition itself is well behaved - for example, no uses of global variables during evaluation of the invariant expression). This goal is not completely achieved; such leaking is possible but, importantly, it requires assistance (deliberate or not) of some form from the package that declares the invariant-bearing private type (or a child unit thereof); a client of a well-crafted package cannot use these holes to obtain an invariant-violating value without help. 23.a/5

{*AI12-0210-1*} The list of known techniques whereby this kind of leak can occur (ignoring things like erroneous execution and the various forms of unchecked type conversion) consists of: 23.b/5

- A boundary entity might assign an invariant-violating value to a global variable that is visible to client code. 23.c/5
- {*AI12-0149-1*} Invariant checks on subprogram return are not performed on objects that are accessible only through access-valued components of other objects. This can only cause a leak if there is a type with access-valued components that is used as a parameter or result type of a boundary entity. For a type *T* that has a type invariant, avoiding the declaration of types with access-valued components designating objects with parts of *T* in the package that contains *T* is sufficient to prevent this leak. 23.d/5
- A client could call through an access-to-subprogram value and reach a subprogram body that has visibility on the full declaration of a type; no invariant checks will be performed if the designated subprogram is not itself a boundary subprogram. This leak can only happen if an access-to-subprogram value of a subprogram that is not visible to clients is passed out to clients. 23.e/5
- {*AI12-0167-1*} {*AI12-0191-1*} Invariant checks are only performed for parts of the nominal type for tagged parameters and function results. This means that components of extensions are not checked (these would be very expensive to check as any tagged type *might* have such an extension in the future, even though that would be very unlikely). For this leak to occur for a type *T* that has a type invariant, the body of a boundary 23.f/5

entity of *T* needs to have visibility on a type extension that has components of *T* or access-to-*T* and also has an ancestor type (or class) as a parameter or result of the subprogram.

23.g/5 - {*AI12-0338-1*} Invariant checks are not performed for parts of incomplete types when the completion is not available. For this leak to occur for a type *T* that has a type invariant and is declared in a package *P*, one has to use a **limited with** on a package that has *P* in its semantic closure, and then use a type from that package as a parameter or result of a boundary subprogram for *T* (or as the designated type of a parameter or result of such a subprogram).

23.h/5 - {*AI12-0210-1*} Consider a package *P* which declares an invariant-bearing private type *T* and a generic package *P.G1*, which in turn declares another generic package *P.G1.G2*. Outside of package *P*, there are declarations of an instantiation *I1* of *P.G1* and an instantiation *I2* of *I1.G2*. *I2* can declare visible subprograms whose bodies see the full view of *T* and yet these subprograms are not boundary subprograms (because the generic *I1.G2* is not declared within the immediate scope of *T* - *G1.G2* is, but that's irrelevant). So a call to one of these subprograms from outside of *P* could yield an invariant-violating value. So long as a nested generic of a nested generic unit of *P* is not declared, no such leaks are possible.

23.i/5 {*AI12-0210-1*} All of these leaks require cooperation of some form (as detailed above) from within the immediate scope of the invariant-bearing type.

23.j/5 **Implementation Note:** The implementation might want to produce a warning if a private extension has an ancestor type that is a visible extension, and an invariant expression depends on the value of one of the components from a visible extension part.

24/3 NOTE {*AI05-0250-1*} {*AI05-0269-1*} For a call of a primitive subprogram of type *NT* that is inherited from type *T*, the specified checks of the specific invariants of both the types *NT* and *T* are performed. For a call of a primitive subprogram of type *NT* that is overridden for type *NT*, the specified checks of the specific invariants of only type *NT* are performed.

24.a/3 **Proof:** This follows from the definition of a call on an inherited subprogram as view conversions of the parameters of the type and a call to the original subprogram (see 3.4), along with the normal invariant checking rules. In particular, the call to the original subprogram takes care of any checks needed on type *T*, and the checks required on view conversions take care of any checks needed on type *NT*, specifically on **in out** and **out** parameters. We require this in order that the semantics of an explicitly defined wrapper that does nothing but call the original subprogram is the same as that of an inherited subprogram.

Examples

25/5 *{AI12-0312-1} {AI12-0429-1} Example of a work scheduler where only urgent work can be scheduled for weekends:*

26/5
```
package Work_Orders is
```

27/5
```
   -- See 3.5.1 for type declarations of Level, Day, and Weekday
```

28/5
```
   type Work_Order is private with
     Type_Invariant => Day_Scheduled (Work_Order) in Weekday
                       or else Priority (Work_Order) = Urgent;
```

29/5
```
   function Schedule_Work (Urgency  : in Level;
                           To_Occur : in Day) return Work_Order
     with Pre => Urgency = Urgent or else To_Occur in Weekday;
```

30/5
```
   function Day_Scheduled (Order : in Work_Order) return Day;
```

31/5
```
   function Priority (Order : in Work_Order) return Level;
```

32/5
```
   procedure Change_Priority (Order        : in out Work_Order;
                              New_Priority : in     Level;
                              Changed      : out    Boolean)
      with Post => Changed = (Day_Scheduled(Order) in Weekday
                              or else Priority(Order) = Urgent);
```

33/5
```
private
```

34/5
```
   type Work_Order is record
      Scheduled : Day;
      Urgency   : Level;
   end record;
```

35/5
```
end Work_Orders;
```

36/5
```
package body Work_Orders is
```

```
   function Schedule_Work (Urgency  : in Level;                                    37/5
                           To_Occur : in Day) return Work_Order is
     (Scheduled => To_Occur, Urgency => Urgency);
   function Day_Scheduled (Order : in Work_Order) return Day is                    38/5
     (Order.Scheduled);
   function Priority (Order : in Work_Order) return Level is                       39/5
     (Order.Urgency);
   procedure Change_Priority (Order        : in out Work_Order;                    40/5
                              New_Priority : in     Level;
                              Changed      : out    Boolean) is
   begin
      -- Ensure type invariant is not violated
      if Order.Urgency = Urgent or else (Order.Scheduled in Weekday) then
         Changed := True;
         Order.Urgency := New_Priority;
      else
         Changed := False;
      end if;
   end Change_Priority;
end Work_Orders;                                                                   41/5
```

Extensions to Ada 2005

{*AI05-0146-1*} {*AI05-0247-1*} {*AI05-0250-1*} {*AI05-0289-1*} Type_Invariant aspects are new. 41.a/3

Inconsistencies With Ada 2012

{*AI12-0042-1*} **Corrigendum:** Clarified the definition of when invariant checks occur for inherited subprograms. This might cause checks to be added or removed in some cases. These are all rare cases involving class-wide type invariants and either record extensions or multiple levels of derivation. Additionally, implementations probably make the checks as the intent seems clear, even though the formal language did not include them. So we do not expect this to be a problem in practice. 41.b/4

{*AI12-0042-1*} **Corrigendum:** Added invariant checks for conversions to class-wide types. This might cause an invariant check to fail in some cases where they would not be made in the original definition of Ada 2012. Such cases represent a hole where a value that fails an invariant could "leak out" of a package, and as such will detect far more bugs than it causes. 41.c/4

{*AI12-0044-1*} **Corrigendum:** Removed the invariant check for **in** parameters of functions, so that typical invariants don't cause infinite recursion. This is strictly inconsistent, as the Ada 2012 definition has this check; therefore, programs could depend on Assertion_Error being raised upon the return from some call on a public function. However, as the intent of assertion checking is to uncover bugs, a program that depends on a bug occurring seems very unlikely. 41.d/4

{*AI12-0049-1*} {*AI12-0149-1*} **Corrigendum:** Added an invariant check for deferred constants and for access values returned from functions, so they cannot be used to “leak” values that violate the invariant from a package. This is strictly inconsistent, as the Ada 2012 definition is missing these checks; therefore, programs could depend on using values that violate an invariant outside of the package of definition. These will not raise Assertion_Error in Ada 2012 as defined in the Ada 2012 Reference Manual, but ought to do so (as noted by this change). As these are a violation of the intent of invariants, we think that this change will mainly reveal bugs rather than cause them. 41.e/4

{*AI12-0150-1*} {*AI12-0159-1*} **Corrigendum:** Eliminated unintentional redispatching from class-wide type invariants. This means that a different body might be evaluated for a type invariant check where the value has a different tag than that of the type. The change means that the behavior of Type_Invariant and Type_Invariant'Class will be the same for a particular subprogram, and that the known behavior of the operations can be assumed. We expect that this change will primarily fix bugs, as it will make class-wide type invariants work more like expected. In the case where redispatching is desired, an explicit conversion to a class-wide type can be used. 41.f/4

{*AI12-0199-1*} **Correction:** Class-wide type invariants are no longer checked for abstract types. Thus, a program that previously raised Assertion_Error because of a call to a concrete subprogram of an abstract type will no longer do so. However, programs that depend on assertion failure are likely to be very rare, some explicit conversion to the abstract type is needed to get static binding, and additionally many such checks would call abstract functions (likely causing some compiler failure). As such, this incompatibility most likely will never be seen in practice. 41.g/5

Incompatibilities With Ada 2012

{*AI12-0042-1*} {*AI12-0382-1*} **Corrigendum:** A private operation that is inherited in the visible part of a package to which a class-wide invariant applies now requires overriding. This is a very unlikely situation, and will prevent 41.h/5

problems with invariant checks being added to routines that assume that they don't have them. Note: The original wording was missing the restriction to the visible part of the package, this was added later for Ada 2022.

Extensions to Ada 2012

41.i/4 {*AI12-0041-1*} **Corrigendum:** Class-wide type invariants can now be specified on interfaces as well as private types.

Wording Changes from Ada 2012

41.j/4 {*AI12-0133-1*} **Corrigendum:** Clarified that all objects that are initialized by default should have an invariant check, and added an exception for types with unknown discriminants, as in that case the client cannot declare a default-initialized object. This exception to the check is formally inconsistent, but since it is only removing an assertion failure that occurs where no assertion should be checked anyway (meaning it's more likely to fix a bug than cause one), and programs depending on assertion failure should be very rare outside of test cases, we don't document this as inconsistent.

41.k/5 {*AI12-0075-1*} {*AI12-0191-1*} Defined the term “boundary entity” to separate the static rules from the dynamic rules, and allow rules in other subclauses to reference these rules. No semantic change is intended.

41.l/5 {*AI12-0191-1*} Clarified that invariant checks only apply to parts of the nominal type of objects.

41.m/5 {*AI12-0193-1*} **Correction:** Clarified when type invariant checks happen for protected actions and entry calls.

41.n/5 {*AI12-0338-1*} **Correction:** Clarified that type invariant checks do not occur for parts of incomplete types unless the completion is available.

7.3.3 Default Initial Conditions

1/5 {*AI12-0265-1*} {*AI12-0272-1*} {*AI12-0396-1*} For a private type or private extension (including a generic formal type), the following language-defined assertion aspect may be specified with an aspect_specification (see 13.1.1):

2/5 {*AI12-0265-1*} {*AI12-0419-1*} Default_Initial_Condition

This aspect shall be specified by an expression, called a *default initial condition expression*. Default_Initial_Condition may be specified on a private_type_declaration, a private_extension_declaration, a formal_private_type_definition, or a formal_derived_type_definition. [The Default_Initial_Condition aspect is not inherited, but its effects are additive, as defined below.]

2.a/5 **Aspect Description for Default_Initial_Condition:** A condition that will hold true after the default initialization of an object.

2.b/5 **Term entry: default initial condition** — property that holds for every default-initialized object of a given type

Name Resolution Rules

3/5 {*AI12-0265-1*} The expected type for a default initial condition expression is any boolean type.

4/5 {*AI12-0397-1*} Within a default initial condition expression associated with a declaration for a type T, a name that denotes the declaration is interpreted as a current instance of a notional (nonabstract) formal derived type NT with ancestor T, that has directly visible primitive operations.

4.a/5 **Reason:** This is analogous to the rule for Post'Class (see 6.1.1) and ensures that the expression is well-defined for any descendant of type T.

4.b/5 **Ramification:** The operations of NT are also nonabstract, so the rule against a call of an abstract subprogram does not trigger for a default initial condition for an abstract type. Note that, presuming T is tagged, it is possible to call class-wide operations of the type T given an object of type NT. Similarly it is possible to explicitly convert an object of type NT to a subtype of T, and pass it to a nonprimitive operation expecting a parameter of type T. [Note that one cannot directly convert to (the first subtype of) T since it represents the current instance of the type within the aspect expression, but one can convert to a subtype of T (including a subtype that matches the first subtype).]

Legality Rules

5/5 {*AI12-0265-1*} The Default_Initial_Condition aspect shall not be specified for a type whose partial view has unknown discriminants[, whether explicitly declared or inherited].

Static Semantics

{*AI12-0265-1*} If the Default_Initial_Condition aspect is specified for a type T, then the default initial condition expression applies to T and to all descendants of T. 6/5

Dynamic Semantics

{*AI12-0265-1*} {*AI12-0397-1*} If one or more default initial condition expressions apply to a [(nonabstract)] type T, then a default initial condition check is performed after successful initialization of an object of type T by default (see 3.3.1). In the case of a controlled type, the check is performed after the call to the type's Initialize procedure (see 7.6). 7/5

Proof: {*AI12-0397-1*} If T is an abstract type, then there will never be an initialization of an object of the type. 7.a/5

{*AI12-0265-1*} If performing checks is required by the Default_Initial_Condition assertion policy (see 11.4.2) in effect at the point of the corresponding aspect_specification applicable to a given type, then the respective default initial condition expression is considered enabled. 8/5

{*AI12-0265-1*} {*AI12-0397-1*} The default initial condition check consists of the evaluation of each enabled default initial condition expression that applies to T. Any operations within such an expression that were resolved as primitive operations of the (notional) formal derived type NT, are in the evaluation of the expression resolved as for a formal derived type in an instance with T as the actual type for NT (see 12.5.1). These evaluations, if there are more than one, are performed in an arbitrary order. If any of these evaluate to False, Assertions.Assertion_Error is raised at the point of the object initialization. 9/5

Ramification: {*AI12-0397-1*} Just as is true for a formal derived type (see 12.5.1), for a tagged type T, the controlling tag of a call on a primitive of NT will cause the body of the corresponding primitive of T to be executed. For an untagged type T, invoking a primitive of NT will cause the body of the operation of the type where the aspect originated to be executed, with conversions performed as for an inherited subprogram. 9.a/5

{*AI12-0272-1*} [For a generic formal type T, default initial condition checks performed are as determined by the actual type, along with any default initial condition of the formal type itself.] 10/5

Proof: This follows from the general dynamic semantics rules given above, but we mention it explicitly so that there can be no doubt that it is intended. 10.a/5

Implementation Permissions

{*AI12-0332-1*} Implementations may extend the syntax or semantics of the Default_Initial_Condition aspect in an implementation-defined manner. 11/5

Implementation defined: Any extensions of the Default_Initial_Condition aspect. 11.a/5

Reason: This is intended to allow preexisting usages from SPARK 2014 to remain acceptable in conforming implementations, as well as to provide future flexibility. Note the word “extend” in this permission; we expect that any aspect usage that conforms with the (other) rules of this subclause will be accepted by any Ada implementation, regardless of any implementation-defined extensions. 11.b/5

NOTE {*AI12-0312-1*} For an example of the use of this aspect, see the Vector container definition in A.18.2. 12/5

Extensions to Ada 2012

{*AI12-0265-1*} {*AI12-0272-1*} {*AI12-0397-1*} Aspect Default_Initial_Condition is new. 12.a/5

7.3.4 Stable Properties of a Type

{*AI12-0187-1*} {*AI12-0324-1*} Certain characteristics of an object of a given type are unchanged by most of the primitive operations of the type. Such characteristics are called *stable properties* of the type. 1/5

Term entry: stable property — characteristic associated with objects of a given type that is preserved by many of the primitive operations of the type 1.a/5

Static Semantics

2/5 {*AI12-0187-1*} A *property function* of type *T* is a function with a single parameter of type *T* or of a class-wide type that covers *T*.

2.a/5 **Reason:** This provides restrictions on name resolution so overloaded functions can be used as a stable property function.

3/5 {*AI12-0285-1*} {*AI12-0324-1*} {*AI12-0388-1*} A *type property aspect definition* is a list of names written in the syntax of a positional_array_aggregate. A *subprogram property aspect definition* is a list of names, each optionally preceded by reserved word **not**, also written in the syntax of a positional_array_aggregate.

3.a/5 **To be honest:** A single name would technically be a parenthesized expression rather than an aggregate; we mean to include that here. We say "syntax of a positional_array_aggregate" to hopefully clarify that the specification is in no other way an actual aggregate.

4/5 {*AI12-0187-1*} {*AI12-0272-1*} For a nonformal private type, nonformal private extension, or full type that does not have a partial view, the following language-defined aspects may be specified with an aspect_specification (see 13.1.1):

5/5 {*AI12-0187-1*} {*AI12-0285-1*} {*AI12-0405-1*} Stable_Properties

This aspect shall be specified by a type property aspect definition; each name shall statically denote a single property function of the type. This aspect defines the *specific stable property functions* of the associated type.

5.a/5 **Discussion:** We do not allow this aspect on generic formal types, as it is only meaningful for primitive subprograms and generic formal types have no such subprograms.

5.b/5 **Ramification:** Class-wide aspects are only allowed on tagged types (see 13.1.1), so we don't say that here.

5.c/5 **Aspect Description for Stable_Properties:** A list of functions describing characteristics that usually are unchanged by primitive operations of the type or an individual primitive subprogram.

6/5 {*AI12-0187-1*} {*AI12-0285-1*} Stable_Properties'Class

This aspect shall be specified by a type property aspect definition; each name shall statically denote a single property function of the type. This aspect defines the *class-wide stable property functions* of the associated type. [Unlike most class-wide aspects, Stable_Properties'Class is not inherited by descendant types and subprograms, but the enhanced class-wide postconditions are inherited in the normal manner.]

6.a/5 **Proof:** Class-wide inheritance has to be explicitly defined. Here we are not making such a definition, so there is no inheritance. 6.1.1 defines the inheritance of class-wide postconditions.

6.b/5 **Discussion:** Since class-wide postconditions are inherited by descendants, we don't need the stable property functions to be inherited; if they were inherited, we'd be duplicating the checks, which we don't want.

6.c/5 **Ramification:** Class-wide aspects are only allowed on primitive subprograms of tagged types (see 13.1.1), so we don't say that here.

6.d/5 **Aspect Description for Stable_Properties'Class:** A list of functions describing characteristics that usually are unchanged by primitive operations of a class of types or a primitive subprogram for such a class.

7/5 {*AI12-0405-1*} The specific and class-wide stable properties of a type together comprise the stable properties of the type.

8/5 {*AI12-0187-1*} For a primitive subprogram, the following language-defined aspects may be specified with an aspect_specification (see 13.1.1):

9/5 {*AI12-0187-1*} {*AI12-0285-1*} Stable_Properties

This aspect shall be specified by a subprogram property aspect definition; each name shall statically denote a single property function of a type for which the associated subprogram is primitive.

{*AI12-0187-1*} {*AI12-0285-1*} Stable_Properties'Class 10/5

This aspect shall be specified by a subprogram property aspect definition; each name shall statically denote a single property function of a tagged type for which the associated subprogram is primitive. [Unlike most class-wide aspects, Stable_Properties'Class is not inherited by descendant subprograms, but the enhanced class-wide postconditions are inherited in the normal manner.]

Reason: The subprogram versions of Stable_Properties are provided to allow overriding the stable properties of a type for an individual primitive subprogram. While they can be used even if the type has no stable properties, that is not an intended use (as simply modifying the postcondition directly makes more sense for something that only happens in one place). 10.a/5

Legality Rules

{*AI12-0187-1*} A stable property function of a type *T* shall have a nonlimited return type and shall be: 11/5

- a primitive function with a single parameter of mode **in** of type *T*; or 12/5
- a function that is declared immediately within the declarative region in which an ancestor type of *T* is declared and has a single parameter of mode **in** of a class-wide type that covers *T*. 13/5

{*AI12-0187-1*} {*AI12-0285-1*} In a subprogram property aspect definition for a subprogram *S*: 14/5

- all or none of the items shall be preceded by **not**; 15/5

  Ramification: Mixing **not** functions with regular functions is not allowed. 15.a/5

- any property functions mentioned after **not** shall be a stable property function of a type for which *S* is primitive. 16/5

{*AI12-0405-1*} If a subprogram_renaming_declaration declares a primitive subprogram of a type *T*, then the renamed callable entity shall also be a primitive subprogram of type *T* and the two primitive subprograms shall have the same specific stable property functions and the same class-wide stable property functions (see below). 17/5

Static Semantics

{*AI12-0187-1*} For a primitive subprogram *S* of a type *T*, the specific stable property functions of *S* for type *T* are: 18/5

- if *S* has an aspect Stable_Properties specified that does not include **not**, those functions denoted in the aspect Stable_Properties for *S* that have a parameter of *T* or *T*'Class; 19/5
- if *S* has an aspect Stable_Properties specified that includes **not**, those functions denoted in the aspect Stable_Properties for *T*, excluding those denoted in the aspect Stable_Properties for *S*; 20/5
- if *S* does not have an aspect Stable_Properties, those functions denoted in the aspect Stable_Properties for *T*, if any. 21/5

  Discussion: A primitive subprogram can be primitive for more than one type, and thus there can be more than one such set of specific stable properties for a subprogram. Thus we say "specific stable property functions for subprogram *S* for type *T*". 21.a/5

{*AI12-0187-1*} A similar definition applies for class-wide stable property functions by replacing aspect Stable_Properties with aspect Stable_Properties'Class in the above definition. 22/5

{*AI12-0187-1*} The *explicit* specific postcondition expression for a subprogram *S* is the expression directly specified for *S* with the Post aspect. Similarly, the *explicit* class-wide postcondition expression for a subprogram *S* is the expression directly specified for *S* with the Post'Class aspect. 23/5

{*AI12-0187-1*} For a primitive subprogram *S* of a type *T* that has a parameter *P* of type *T*, the parameter is *excluded from stable property checks* if: 24/5

- *S* is a stable property function of *T*; 25/5

25.a/5 **Reason:** This prevents possible infinite recursion, where the postcondition calls the function itself (directly or indirectly).

26/5 - *P* has mode **out**;

26.a/5 **Reason:** A parameter of mode **out** doesn't necessarily have a defined input value, so there is no old value to compare with. Ideally, the postcondition will include expressions defining the values of the stable properties after the call, but we do not try to ensure this.

27/5 - the Global aspect of *S* is **null**, and *P* has mode **in** and the mode is not overridden by a global aspect.

27.a/5 **Reason:** An **in** parameter of a Global => **null** subprogram cannot be modified, even if it has indirect parts, without violating the Global aspect. Thus, there is no need to assert that the properties don't change.

28/5 {*AI12-0187-1*} {*AI12-0324-1*} {*AI12-0405-1*} For every primitive subprogram *S* of a type *T* that is not an abstract subprogram or null procedure, the specific postcondition expression of *S* is modified to include expressions of the form `F(P) = F(P)'Old`, all **and**ed with each other and any explicit specific postcondition expression, with one such equality included for each specific stable property function *F* of *S* for type *T* that does not occur in the explicit specific postcondition expression of *S*, and *P* is each parameter of *S* that has type *T* and is not excluded from stable property checks. The resulting specific postcondition expression of *S* is used in place of the explicit specific postcondition expression of *S* [when interpreting the meaning of the postcondition as defined in 6.1.1].

28.a/5 **Ramification:** There is one `F(P) = F(P)'Old` subexpression for every combination of a specific stable property function of type *T* and a parameter of type *T*. For instance, if there are three specific stable property functions for type *T* and two parameters of type *T*, then there are six such subexpressions appended to the postcondition.

28.b/5 The resulting specific postcondition is evaluated as described in 6.1.1. One hopes that compilers can be smart enough to prove that many of these added postcondition subexpressions cannot fail, but that is not required here.

28.c/5 **Reason:** Null procedures and abstract subprograms are excluded as they do not allow specific postconditions. Moreover, for null procedures, static analysis tools can be certain that their parameters aren't modified so there is no need to assert that the properties don't change. Abstract subprograms cannot be directly called.

29/5 {*AI12-0187-1*} {*AI12-0324-1*} {*AI12-0405-1*} For every primitive subprogram *S* of a type *T*, the class-wide postcondition expression of *S* is modified to include expressions of the form `F(P) = F(P)'Old`, all **and**ed with each other and any explicit class-wide postcondition expression, with one such equality included for each class-wide stable property function *F* of *S* for type *T* that does not occur in any class-wide postcondition expression that applies to *S*, and *P* is each parameter of *S* that has type *T* and is not excluded from stable property checks. The resulting class-wide postcondition expression of *S* is used in place of the explicit class-wide postcondition expression of *S* [when interpreting the meaning of the postcondition as defined in 6.1.1].

29.a/5 **Reason:** We suppress stable property expressions if the property function appears in the explicit class-wide postcondition, or in any inherited class-wide postconditions. If we didn't do that, we could have conflicting requirements in an inherited postcondition and the current one. We also avoid redundant property checks.

29.b/5 **Ramification:** The resulting class-wide postcondition is evaluated as described in 6.1.1. In particular, the enhanced class-wide postcondition *is* the class-wide postcondition for *S*, and therefore inherited postconditions include any stable property expressions for *S*.

29.c/5 {*AI12-0405-1*} In the case of a derived type *T*, when the preceding rules refer to "every primitive subprogram *S* of a type *T*", the referenced set of subprograms includes any inherited subprograms.

30/5 {*AI12-0405-1*} The equality operation that is used in the aforementioned equality expressions is as described in the case of an individual membership test whose membership_choice is a *choice*_simple_expression (see 4.5.2).

31/5 {*AI12-0405-1*} The Post expression additions described above are enabled or disabled depending on the Post assertion policy that is in effect at the point of declaration of the subprogram *S*. A similar rule applies to the Post'Class expression additions.

32/5 NOTE {*AI12-0112-1*} For an example of the use of these aspects, see the Vector container definition in A.18.2.

Extensions to Ada 2012

{*AI12-0187-1*} {*AI12-0285-1*} {*AI12-0405-1*} These aspects are new. 32.a/5

7.4 Deferred Constants

[Deferred constant declarations may be used to declare constants in the visible part of a package, but with the value of the constant given in the private part. They may also be used to declare constants imported from other languages (see Annex B).] 1

Legality Rules

{*AI05-0229-1*} {*AI05-0269-1*} [A *deferred constant declaration* is an object_declaration with the reserved word **constant** but no initialization expression.] The constant declared by a deferred constant declaration is called a *deferred constant*. [Unless the Import aspect (see B.1) is True for a deferred constant declaration, the] deferred constant declaration requires a completion, which shall be a full constant declaration (called the *full declaration* of the deferred constant). 2/3

Proof: The first sentence is redundant, as it is stated officially in 3.3.1. 2.a

{*AI05-0229-1*} {*AI05-0269-1*} The first part of the last sentence is redundant, as no imported entity may have a completion, as stated in B.1. 2.b/3

A deferred constant declaration that is completed by a full constant declaration shall occur immediately within the visible part of a package_specification. For this case, the following additional rules apply to the corresponding full declaration: 3

- The full declaration shall occur immediately within the private part of the same package; 4
- {*AI95-00385-01*} The deferred and full constants shall have the same type, or shall have statically matching anonymous access subtypes; 5/2

  Ramification: {*AI95-00385-01*} This implies that both the deferred declaration and the full declaration have to have a subtype_indication or access_definition rather than an array_type_definition, because each array_type_definition would define a new type. 5.a/2
- {*AI95-00385-01*} {*AI05-0062-1*} {*AI05-0262-1*} If the deferred constant declaration includes a subtype_indication *S* that defines a constrained subtype, then the constraint defined by the subtype_indication in the full declaration shall match the constraint defined by *S* statically.[On the other hand, if the subtype of the deferred constant is unconstrained, then the full declaration is still allowed to impose a constraint. The constant itself will be constrained, like all constants;] 6/3
- {*AI95-00231-01*} If the deferred constant declaration includes the reserved word **aliased**, then the full declaration shall also; 7/2

  Ramification: On the other hand, the full constant can be aliased even if the deferred constant is not. 7.a
- {*AI95-00231-01*} If the subtype of the deferred constant declaration excludes null, the subtype of the full declaration shall also exclude null. 7.1/2

  Ramification: On the other hand, the full constant can exclude null even if the deferred constant does not. But that can only happen for a subtype_indication, as anonymous access types are required to statically match (which includes any null_exclusion). 7.a.1/2

{*AI05-0229-1*} {*AI12-0444-1*} [A deferred constant declaration for which the Import aspect is True can appear anywhere that an object_declaration is allowed, and has no full constant declaration.] 8/5

{*AI95-00256-01*} The completion of a deferred constant declaration shall occur before the constant is frozen (see 13.14). 9/2

Dynamic Semantics

10/3 {*AI05-0004-1*} The elaboration of a deferred constant declaration elaborates the subtype_indication, access_definition, or (only allowed in the case of an imported constant) the array_type_definition.

10.a/3 **Ramification:** {*AI05-0004-1*} For nonimported constants, these elaborations cannot require any code or checks for a legal program, because the given subtype_indication has to be indefinite or statically match that of the full constant, meaning that either it is a subtype_mark or it has static constraints. If the deferred constant instead has an access_definition, the designated subtype must be a subtype_mark. We still say that these are elaborated, however, because part of elaboration is creating the type, which is clearly needed for access_definitions. (A deferred constant and its full constant have different types when they are specified by an access_definition, although there is no visible effect of these types being different as neither can be named.)

11 NOTE The full constant declaration for a deferred constant that is of a given private type or private extension is not allowed before the corresponding full_type_declaration. This is a consequence of the freezing rules for types (see 13.14).

11.a **Ramification:** Multiple or single declarations are allowed for the deferred and the full declarations, provided that the equivalent single declarations would be allowed.

11.b Deferred constant declarations are useful for declaring constants of private views, and types with components of private views. They are also useful for declaring access-to-constant objects that designate variables declared in the private part of a package.

Examples

12 *Examples of deferred constant declarations:*

13
```
Null_Key : constant Key;       -- see 7.3.1
```

14/3
```
{AI05-0229-1} CPU_Identifier : constant String(1..8)
   with Import => True, Convention => Assembler, Link_Name => "CPU_ID";
                               -- see B.1
```

Extensions to Ada 83

14.a In Ada 83, a deferred constant is required to be of a private type declared in the same visible part. This restriction is removed for Ada 95; deferred constants can be of any type.

14.b In Ada 83, a deferred constant declaration was not permitted to include a constraint, nor the reserved word **aliased**.

14.c In Ada 83, the rules required conformance of type marks; here we require static matching of subtypes if the deferred constant is constrained.

14.d A deferred constant declaration can be completed with a pragma Import. Such a deferred constant declaration need not be within a package_specification.

14.e The rules for too-early uses of deferred constants are modified in Ada 95 to allow more cases, and catch all errors at compile time. This change is necessary in order to allow deferred constants of a tagged type without violating the principle that for a dispatching call, there is always an implementation to dispatch to. It has the beneficial side effect of catching some Ada-83-erroneous programs at compile time. The new rule fits in well with the new freezing-point rules. Furthermore, we are trying to convert undefined-value problems into bounded errors, and we were having trouble for the case of deferred constants. Furthermore, uninitialized deferred constants cause trouble for the shared variable / tasking rules, since they are really variable, even though they purport to be constant. In Ada 95, they cannot be touched until they become constant.

14.f Note that we do not consider this change to be an upward incompatibility, because it merely changes an erroneous execution in Ada 83 into a compile-time error.

14.g The Ada 83 semantics are unclear in the case where the full view turns out to be an access type. It is a goal of the language design to prevent uninitialized access objects. One wonders if the implementation is required to initialize the deferred constant to null, and then initialize it (again!) to its real value. In Ada 95, the problem goes away.

Wording Changes from Ada 83

14.h/3 {*AI05-0299-1*} Since deferred constants can now be of a nonprivate type, we have made this a stand-alone subclause, rather than a subclause of 7.3, “Private Types and Private Extensions”.

14.i Deferred constant declarations used to have their own syntax, but now they are simply a special case of object_declarations.

Extensions to Ada 95

14.j/2 {*AI95-00385-01*} Deferred constants were enhanced to allow the use of anonymous access types in them.

Wording Changes from Ada 95

{*AI95-00231-01*} Added matching rules for subtypes that exclude null. 14.k/2

Wording Changes from Ada 2005

{*AI05-0062-1*} **Correction:** Corrected rules so that the intent that a full constant may have a null exclusion even if the deferred constant does not is actually met. 14.l/3

7.5 Limited Types

{*AI95-00287-01*} [A limited type is (a view of) a type for which copying (such as for an assignment_statement) is not allowed. A nonlimited type is a (view of a) type for which copying is allowed.] 1/2

Discussion: The concept of the *value* of a limited type is difficult to define, since the abstract value of a limited type often extends beyond its physical representation. In some sense, values of a limited type cannot be divorced from their object. The value *is* the object. 1.a

{*AI95-00318-02*} In Ada 83, in the two places where limited types were defined by the language, namely tasks and files, an implicit level of indirection was implied by the semantics to avoid the separation of the value from an associated object. In Ada 95, most limited types are passed by reference, and even return-ed by reference. In Ada 2005, most limited types are built-in-place upon return, rather than returned by reference. Thus the object "identity" is part of the logical value of most limited types. 1.b/2

To be honest: {*AI95-00287-01*} {*AI95-00419-01*} For a limited partial view whose full view is nonlimited, copying is possible on parameter passing and function return. To prevent any copying whatsoever, one should make both the partial *and* full views limited. 1.c/2

Term entry: limited type — type for which copying (such as in an assignment_statement) is not allowed
Note: All types are either limited types or nonlimited types. 1.d/5

Term entry: nonlimited type — type for which copying is allowed 1.e/5

Legality Rules

{*AI95-00419-01*} If a tagged record type has any limited components, then the reserved word **limited** shall appear in its record_type_definition. [If the reserved word **limited** appears in the definition of a derived_type_definition, its parent type and any progenitor interfaces shall be limited.] 2/2

Proof: {*AI95-00419-01*} {*AI12-0005-1*} The rule about the parent type being required to be limited can be found in 3.4. Rules about progenitor interfaces can be found in 3.9.4; specifically, a nonlimited interface can appear only on a nonlimited type. We repeat these rules here to gather these scattered rules in one obvious place. 2.a.1/5

Reason: This prevents tagged limited types from becoming nonlimited. Otherwise, the following could happen: 2.a

```
package P is                                                    2.b
    type T is limited private;
    type R is tagged
        record -- Illegal!
                -- This should say "limited record".
            X : T;
        end record;
private
    type T is new Integer; -- R becomes nonlimited here.
end P;

package Q is                                                    2.c/2
    type R2 is new R with
        record
            Y : Some_Task_Type;
        end record;
end Q;
```

{*AI95-00230-01*} If the above were legal, then assignment would be defined for R'Class in the body of P, which is bad news, given the task. 2.d/2

{*AI95-00287-01*} {*AI95-00318-02*} {*AI05-0147-1*} {*AI12-0172-1*} {*AI12-0236-1*} {*AI12-0317-1*} In the following contexts, an expression of a limited type is permitted only if each of its operative constituents is newly constructed (see 4.4): 2.1/5

2.2/2 • the initialization expression of an object_declaration (see 3.3.1)

2.3/2 • the default_expression of a component_declaration (see 3.8)

2.4/2 • the expression of a record_component_association (see 4.3.1)

2.5/2 • the expression for an ancestor_part of an extension_aggregate (see 4.3.2)

2.6/2 • an expression of a positional_array_aggregate or the expression of an array_component_association (see 4.3.3)

2.7/5 • {*AI12-0127-1*} the *base*_expression of a record_delta_aggregate (see 4.3.4)

2.d.1/5 **Ramification:** {*AI12-0127-1*} We don't need to mention the *base*_expression of an array_delta_aggregate here, as its type cannot be limited (see 4.3.4), and thus neither can its *base*_expression. Similarly, we do not need any rules for components of a delta_aggregate nor the elements of a container_aggregate, as neither are allowed to be limited (see 4.3.4 and 4.3.5)

2.8/5 • the qualified_expression of an initialized allocator (see 4.8)

2.9/5 • the expression of a return statement (see 6.5)

2.10/5 • {*AI05-0177-1*} {*AI12-0157-1*} the return expression of an expression function (see 6.8)

2.11/5 • the default_expression or actual parameter for a formal object of mode **in** (see 12.4)

2.e/2 **Discussion:** All of these contexts normally require copying; by restricting the uses as above, we can require the new object to be built-in-place.

Static Semantics

3/3 {*AI95-00419-01*} {*AI05-0178-1*} A view of a type is *limited* if it is one of the following:

4/2 • {*AI95-00411-01*} {*AI95-00419-01*} a type with the reserved word **limited**, **synchronized**, **task**, or **protected** in its definition;

4.a **Ramification:** Note that there is always a "definition", conceptually, even if there is no syntactic category called "..._definition".

4.b/2 {*AI95-00419-01*} This includes interfaces of the above kinds, derived types with the reserved word **limited**, as well as task and protected types.

5/3 • {*AI95-00419-01*} {*AI05-0087-1*} a class-wide type whose specific type is limited;

6/2 • {*AI95-00419-01*} a composite type with a limited component;

6.1/3 • {*AI05-0178-1*} an incomplete view;

6.2/2 • {*AI95-00419-01*} a derived type whose parent is limited and is not an interface.

6.a/2 **Ramification:** {*AI95-00419-01*} Limitedness is not inherited from interfaces; it must be explicitly specified when the parent is an interface.

6.b/2 **To be honest:** {*AI95-00419-01*} A derived type can become nonlimited if **limited** does not appear and the derivation takes place in the visible part of a child package, and the parent type is nonlimited as viewed from the private part or body of the child package.

6.c/2 **Reason:** {*AI95-00419-01*} We considered a rule where limitedness was always inherited from the parent for derived types, but in the case of a type whose parent is an interface, this meant that the first interface is treated differently than other interfaces. It also would have forced users to declare dummy nonlimited interfaces just to get the limitedness right. We also considered a syntax like **not limited** to specify nonlimitedness when the parent was limited, but that was unsavory. The rule given is more uniform and simpler to understand.

6.d/2 {*AI95-00419-01*} The rules for interfaces are asymmetrical, but the language is not: if the parent interface is limited, the presence of the word **limited** determines the limitedness, and nonlimited progenitors are illegal by the rules in 3.9.4 if **limited** is present. If the parent interface is nonlimited, the word **limited** is illegal by the rules in 3.4. The net effect is that the order of the interfaces doesn't matter.

7 Otherwise, the type is nonlimited.

8 [There are no predefined equality operators for a limited type.]

{*AI05-0052-1*} A type is *immutably limited* if it is one of the following: 8.1/3

- An explicitly limited record type; 8.2/3
- {*AI05-0217-1*} A record extension with the reserved word **limited**; 8.3/3
- A nonformal limited private type that is tagged or has at least one access discriminant with a default_expression; 8.4/3

  Reason: The full type in both of these cases must necessarily be immutably limited. We need to include private types as much as possible so that we aren't unintentionally discouraging the use of private types. 8.a/3
- A task type, a protected type, or a synchronized interface; 8.5/3
- A type derived from an immutably limited type. 8.6/3

  Discussion: An immutably limited type is a type that cannot become nonlimited subsequently in a private part or in a child unit. If a view of the type makes it immutably limited, then no copying (assignment) operations are ever available for objects of the type. This allows other properties; for instance, it is safe for such objects to have access discriminants that have defaults or designate other limited objects. 8.b/3

  Ramification: A nonsynchronized limited interface type is not immutably limited; a type derived from it can be nonlimited. 8.c/3

{*AI05-0052-1*} A descendant of a generic formal limited private type is presumed to be immutably limited except within the body of a generic unit or a body declared within the declarative region of a generic unit, if the formal type is declared within the formal part of the generic unit. 8.7/3

Ramification: In an instance, a type is descended from the actual type corresponding to the formal, and all rules are rechecked in the specification. Bodies are excepted so that we assume the worst there; the complex wording is required to handle children of generics and unrelated bodies properly. 8.d/3

NOTE 1 {*AI95-00287-01*} {*AI95-00318-02*} {*AI05-0067-1*} {*AI12-0442-1*} While it is allowed to write initializations of limited objects, such initializations never copy a limited object. The source of such an assignment operation will be an aggregate or function_call, and such aggregates and function_calls will be built directly in the target object (see 7.6). 9/5

To be honest: This isn't quite true if the type can become nonlimited (see below); function_calls only are required to be build-in-place for “really” limited types. 9.a/2

Paragraphs 10 through 15 were deleted.

NOTE 2 As illustrated in 7.3.1, an untagged limited type can become nonlimited under certain circumstances. 16

Ramification: Limited private types do not become nonlimited; instead, their full view can be nonlimited, which has a similar effect. 16.a

It is important to remember that a single nonprivate type can be both limited and nonlimited in different parts of its scope. In other words, “limited” is a property that depends on where you are in the scope of the type. We don't call this a “view property” because there is no particular declaration to declare the nonlimited view. 16.b

Tagged types never become nonlimited. 16.c

Examples

Example of a package with a limited type: 17

```
package IO_Package is                                                    18
   type File_Name is limited private;

   procedure Open (F : in out File_Name);                                19
   procedure Close(F : in out File_Name);
   procedure Read (F : in File_Name; Item : out Integer);
   procedure Write(F : in File_Name; Item : in  Integer);
private
   type File_Name is
      limited record
         Internal_Name : Integer := 0;
      end record;
end IO_Package;
```

20
```
package body IO_Package is
   Limit : constant := 200;
   type File_Descriptor is record  ...  end record;
   Directory : array (1 .. Limit) of File_Descriptor;
   ...
   procedure Open (F : in out File_Name) is  ...  end;
   procedure Close(F : in out File_Name) is  ...  end;
   procedure Read (F : in File_Name; Item : out Integer) is ... end;
   procedure Write(F : in File_Name; Item : in  Integer) is ... end;
begin
   ...
end IO_Package;
```

21/5 {*AI12-0440-1*} {*AI12-0452-1*} In the example, an outside subprogram making use of IO_Package can obtain a file name by calling Open and later use it in calls to Read and Write. Thus, outside the package, a file name obtained from Open acts as a kind of password; its internal properties (such as containing a numeric value) are not known and no other operations (such as addition or comparison of internal names) can be performed on a file name. Most importantly, clients of the package cannot make copies of objects of type File_Name.

22/5 {*AI12-0442-1*} {*AI12-0452-1*} This example is characteristic of any case where complete control over the operations of a type is desired. Such packages serve a dual purpose. They prevent a user from making use of the internal structure of the type. They also implement the notion of an encapsulated data type where the only operations on the type are those given in the package specification.

23/5 {*AI12-0442-1*} {*AI12-0452-1*} The fact that the full view of File_Name is explicitly declared **limited** means that parameter passing will always be by reference and function results will always be built directly in the result object (see 6.2 and 6.5).

Extensions to Ada 83

26.a The restrictions in RM83-7.4.4(4), which disallowed **out** parameters of limited types in certain cases, are removed.

Wording Changes from Ada 83

26.b/3 {*AI05-0299-1*} Since limitedness and privateness are orthogonal in Ada 95 (and to some extent in Ada 83), this is now its own subclause rather than being a subclause of 7.3, “Private Types and Private Extensions”.

Extensions to Ada 95

26.c/2 {*AI95-00287-01*} {*AI95-00318-02*} Limited types now have an assignment operation, but its use is restricted such that all uses are build-in-place. This is accomplished by restricting uses to aggregates and function_calls. Aggregates were not allowed to have a limited type in Ada 95, which causes a compatibility issue discussed in 4.3, “Aggregates”. Compatibility issues with return statements for limited function_calls are discussed in 6.5, “Return Statements”.

Wording Changes from Ada 95

26.d/2 {*AI95-00411-01*} {*AI95-00419-01*} Rewrote the definition of limited to ensure that interfaces are covered, but that limitedness is not inherited from interfaces. Derived types that explicitly include **limited** are now also covered.

Wording Changes from Ada 2005

26.e/3 {*AI05-0052-1*} {*AI05-0217-1*} **Correction:** Added a definition for immutably limited types, so that the fairly complex definition does not need to be repeated in rules elsewhere in the Reference Manual.

26.f/3 {*AI05-0067-1*} {*AI05-0299-1*} **Correction:** The built-in-place rules are consolidated in 7.6, and thus they are removed from this subclause.

26.g/3 {*AI05-0087-1*} **Correction:** Fixed an oversight: class-wide types were never defined to be limited, even if their associated specific type is. It is thought that this oversight was never implemented incorrectly by any compiler, thus we have not classified it as an incompatibility.

26.h/3 {*AI05-0147-1*} Allowed conditional_expressions in limited constructor contexts — we want to treat these as closely to parentheses as possible.

26.i/3 {*AI05-0177-1*} Added wording so that expression functions can return limited entities.

{*AI05-0178-1*} **Correction:** Added incomplete views to the list of reasons for a view of a type to be limited. This is not a change as the definition already was in 3.10.1. But it is much better to have all of the reasons for limitedness together. 26.j/3

Extensions to Ada 2012

{*AI12-0172-1*} **Correction:** A raise_expression can be used in an expression used in a limited context. This is harmless (no object will be created if an exception is raised instead), useful, and thus appears to have been an omission when raise_expressions were added to the language. 26.k/5

Wording Changes from Ada 2012

{*AI12-0127-1*} Added the *base*_expression of a delta_aggregate as a limited context. 26.l/5

7.6 Assignment and Finalization

[Three kinds of actions are fundamental to the manipulation of objects: initialization, finalization, and assignment. Every object is initialized, either explicitly or by default, after being created (for example, by an object_declaration or allocator). Every object is finalized before being destroyed (for example, by leaving a subprogram_body containing an object_declaration, or by a call to an instance of Unchecked_Deallocation). An assignment operation is used as part of assignment_statements, explicit initialization, parameter passing, and other operations. 1

Default definitions for these three fundamental operations are provided by the language, but a *controlled* type gives the user additional control over parts of these operations. In particular, the user can define, for a controlled type, an Initialize procedure which is invoked immediately after the normal default initialization of a controlled object, a Finalize procedure which is invoked immediately before finalization of any of the components of a controlled object, and an Adjust procedure which is invoked as the last step of an assignment to a (nonlimited) controlled object.] 2

Term entry: controlled type — type that supports user-defined assignment and finalization
Note: Objects are always finalized before being destroyed. 2.a/5

Ramification: {*AI95-00114-01*} {*AI95-00287-01*} Here's the basic idea of initialization, value adjustment, and finalization, whether or not user defined: When an object is created, if it is explicitly assigned an initial value, the object is either built-in-place from an aggregate or function call (in which case neither Adjust nor Initialize is applied), or the assignment copies and adjusts the initial value. Otherwise, Initialize is applied to it (except in the case of an aggregate as a whole). An assignment_statement finalizes the target before copying in and adjusting the new value. Whenever an object goes away, it is finalized. Calls on Initialize and Adjust happen bottom-up; that is, components first, followed by the containing object. Calls on Finalize happen top-down; that is, first the containing object, and then its components. These ordering rules ensure that any components will be in a well-defined state when Initialize, Adjust, or Finalize is applied to the containing object. 2.b/2

Static Semantics

The following language-defined library package exists: 3

```
{8652/0020} {AI95-00126-01} {AI05-0212-1} {AI12-0241-1} package Ada.Finalization    4/5
    with Pure, Nonblocking => False is

{AI95-00161-01} {AI12-0399-1}     type Controlled is abstract tagged private        5/5
       with Preelaborable_Initialization;

{AI95-00348-01}     procedure Initialize (Object : in out Controlled) is null;      6/2
    procedure Adjust     (Object : in out Controlled) is null;
    procedure Finalize   (Object : in out Controlled) is null;

{AI95-00161-01} {AI12-0399-1}     type Limited_Controlled is abstract tagged       7/5
limited private
       with Preelaborable_Initialization;
```

8/2
```
{AI95-00348-01}     procedure Initialize (Object : in out Limited_Controlled) is
null;
    procedure Finalize   (Object : in out Limited_Controlled) is null;
private
    ... -- not specified by the language
end Ada.Finalization;
```

8.a/5 **Reason:** {*AI12-0241-1*} This package must allow blocking (Nonblocking => False) for compatibility. The purpose of this package is to provide a template for overriding user-defined routines; and such routines can only allow blocking if the root type does so. Users can still declare their overridding routines nonblocking if they wish.

9/2 {*AI95-00348-01*} A controlled type is a descendant of Controlled or Limited_Controlled. The predefined "=" operator of type Controlled always returns True, [since this operator is incorporated into the implementation of the predefined equality operator of types derived from Controlled, as explained in 4.5.2.] The type Limited_Controlled is like Controlled, except that it is limited and it lacks the primitive subprogram Adjust.

9.a **Discussion:** We say “nonlimited controlled type” (rather than just “controlled type”;) when we want to talk about descendants of Controlled only.

9.b **Reason:** We considered making Adjust and Finalize abstract. However, a reasonable coding convention is e.g. for Finalize to always call the parent's Finalize after doing whatever work is needed for the extension part. (Unlike CLOS, we have no way to do that automatically in Ada 95.) For this to work, Finalize cannot be abstract. In a generic unit, for a generic formal abstract derived type whose ancestor is Controlled or Limited_Controlled, calling the ancestor's Finalize would be illegal if it were abstract, even though the actual type might have a concrete version.

9.c Types Controlled and Limited_Controlled are abstract, even though they have no abstract primitive subprograms. It is not clear that they need to be abstract, but there seems to be no harm in it, and it might make an implementation's life easier to know that there are no objects of these types — in case the implementation wishes to make them “magic” in some way.

9.d/2 {*AI95-00251-01*} For Ada 2005, we considered making these types interfaces. That would have the advantage of allowing them to be added to existing trees. But that was rejected both because it would cause massive disruptions to existing implementations, and because it would be very incompatible due to the "no hidden interfaces" rule. The latter rule would prevent a tagged private type from being completed with a derivation from Controlled or Limited_Controlled — a very common idiom.

9.1/2 {*AI95-00360-01*} A type is said to *need finalization* if:

9.2/2 - it is a controlled type, a task type or a protected type; or

9.3/3 - {*AI05-0092-1*} it has a component whose type needs finalization; or

9.4/3 - {*AI05-0013-1*} it is a class-wide type; or

9.5/3 - {*AI05-0026-1*} it is a partial view whose full view needs finalization; or

9.6/2 - it is one of a number of language-defined types that are explicitly defined to need finalization.

9.e/2 **Ramification:** The fact that a type needs finalization does not require it to be implemented with a controlled type. It just has to be recognized by the No_Nested_Finalization restriction.

9.f/2 This property is defined for the type, not for a particular view. That's necessary as restrictions look in private parts to enforce their restrictions; the point is to eliminate all controlled parts, not just ones that are visible.

Dynamic Semantics

10/2 {*AI95-00373-01*} During the elaboration or evaluation of a construct that causes an object to be initialized by default, for every controlled subcomponent of the object that is not assigned an initial value (as defined in 3.3.1), Initialize is called on that subcomponent. Similarly, if the object that is initialized by default as a whole is controlled, Initialize is called on the object.

11/2 {*8652/0021*} {*AI95-00182-01*} {*AI95-00373-01*} For an extension_aggregate whose ancestor_part is a subtype_mark denoting a controlled subtype, the Initialize procedure of the ancestor type is called, unless that Initialize procedure is abstract.

Discussion: Example: 11.a

```
type T1 is new Controlled with                                    11.b
    record
        ... -- some components might have defaults
    end record;
type T2 is new Controlled with                                    11.c
    record
        X : T1; -- no default
        Y : T1 := ...; -- default
    end record;
A : T2;                                                           11.d
B : T2 := ...;
```

As part of the elaboration of A's declaration, A.Y is assigned a value; therefore Initialize is not applied to A.Y. Instead, Adjust is applied to A.Y as part of the assignment operation. Initialize is applied to A.X and to A, since those objects are not assigned an initial value. The assignment to A.Y is not considered an assignment to A. 11.e

For the elaboration of B's declaration, Initialize is not called at all. Instead the assignment adjusts B's value; that is, it applies Adjust to B.X, B.Y, and B. 11.f

{*8652/0021*} {*AI95-00182-01*} {*AI95-00373-01*} The ancestor_part of an extension_aggregate, <> in aggregates, and the return object of an extended_return_statement are handled similarly. 11.f.1/2

Initialize and other initialization operations are done in an arbitrary order, except as follows. Initialize is applied to an object after initialization of its subcomponents, if any [(including both implicit initialization and Initialize calls)]. If an object has a component with an access discriminant constrained by a per-object expression, Initialize is applied to this component after any components that do not have such discriminants. For an object with several components with such a discriminant, Initialize is applied to them in order of their component_declarations. For an allocator, any task activations follow all calls on Initialize. 12

Reason: The fact that Initialize is done for subcomponents first allows Initialize for a composite object to refer to its subcomponents knowing they have been properly initialized. 12.a

The fact that Initialize is done for components with access discriminants after other components allows the Initialize operation for a component with a self-referential access discriminant to assume that other components of the enclosing object have already been properly initialized. For multiple such components, it allows some predictability. 12.b

When a target object with any controlled parts is assigned a value, [either when created or in a subsequent assignment_statement,] the *assignment operation* proceeds as follows: 13

- The value of the target becomes the assigned value. 14
- The value of the target is *adjusted.* 15

Ramification: If any parts of the object are controlled, abort is deferred during the assignment operation. 15.a

{*AI05-0067-1*} To adjust the value of a composite object, the values of the components of the object are first adjusted in an arbitrary order, and then, if the object is nonlimited controlled, Adjust is called. Adjusting the value of an elementary object has no effect[, nor does adjusting the value of a composite object with no controlled parts.] 16/3

Ramification: {*AI05-0067-1*} Adjustment is never actually performed for values of an immutably limited type, since all assignment operations for such types are required to be built-in-place. Even so, we still define adjustment for all types in order that the canonical semantics is well-defined. 16.a/3

Reason: {*AI05-0005-1*} The verbiage in the Initialize rule about access discriminants constrained by per-object expressions is not necessary here, since such types are either limited or do not have defaults, so the discriminant can only be changed by an assignment to an outer object. Such an assignment could happen only before any adjustments or (if part of an outer Adjust) only after any inner (component) adjustments have completed. 16.b/3

For an assignment_statement,[after the name and expression have been evaluated, and any conversion (including constraint checking) has been done,] an anonymous object is created, and the value is assigned into it; [that is, the assignment operation is applied]. [(Assignment includes value adjustment.)] The target of the assignment_statement is then finalized. The value of the anonymous object is then assigned into the target of the assignment_statement. Finally, the anonymous object is finalized. [As explained below, 17

the implementation may eliminate the intermediate anonymous object, so this description subsumes the one given in 5.2, “Assignment Statements”.]

17.a **Reason:** An alternative design for user-defined assignment might involve an Assign operation instead of Adjust:

17.b
```
procedure Assign(Target : in out Controlled; Source : in out Controlled);
```

17.c Or perhaps even a syntax like this:

17.d
```
procedure ":="(Target : in out Controlled; Source : in out Controlled);
```

17.e Assign (or ":=") would have the responsibility of doing the copy, as well as whatever else is necessary. This would have the advantage that the Assign operation knows about both the target and the source at the same time — it would be possible to do things like reuse storage belonging to the target, for example, which Adjust cannot do. However, this sort of design would not work in the case of unconstrained discriminated variables, because there is no way to change the discriminants individually. For example:

17.f
```
type Mutable(D : Integer := 0) is
    record
        X : Array_Of_Controlled_Things(1..D);
        case D is
            when 17 => Y : Controlled_Thing;
            when others => null;
        end D;
    end record;
```

17.g An assignment to an unconstrained variable of type Mutable can cause some of the components of X, and the component Y, to appear and/or disappear. There is no way to write the Assign operation to handle this sort of case.

17.h Forbidding such cases is not an option — it would cause generic contract model violations.

17.1/5 {*AI05-0067-1*} {*AI12-0439-1*} When a function call or aggregate is used to initialize an object, the result of the function call or aggregate is an anonymous object, which is assigned into the newly-created object. For such an assignment, the anonymous object may be *built in place*, in which case the assignment does not involve any copying. Under certain circumstances, the anonymous object is required to be built in place. In particular:

17.i/3 **Discussion:** {*AI05-0067-1*} We say assignment to built-in-place objects does not involve copying, which matches the intended implementation (see below). Of course, the implementation can do any copying it likes, if it can make such copying semantically invisible (by patching up access values to point to the copy, and so forth).

17.2/3 - If the full type of any part of the object is immutably limited, the anonymous object is built in place.

17.j/3 **Reason:** {*AI05-0067-1*} We talk about the full types being immutably limited, as this is independent of the view of a type (in the same way that it is for determining the technique of parameter passing). That is, privacy is ignored for this purpose.

17.k/3 {*AI05-0005-1*} {*AI05-0067-1*} For function calls, we only require building in place for immutably limited types. These are the types that would have been return-by-reference types in Ada 95. We limited the requirement because we want to minimize disruption to Ada 95 implementations and users.

17.l/5 **To be honest:** {*AI05-0232-1*} {*AI12-0005-1*} This is a dynamic property and is determined by the specific type of the parts of the actual object. In particular, if a part has a class-wide type, the tag of the object might need to be examined in order to determine if build-in-place is required. However, we expect that most Ada implementations will determine this property at compile-time using some assume-the-worst algorithm in order to choose the appropriate method to implement a given call or aggregate. In addition, there is no attribute or other method for a program to determine if a particular object has this property (or not), so there is no value to a more careful description of this rule.

17.3/3 - In the case of an aggregate, if the full type of any part of the newly-created object is controlled, the anonymous object is built in place.

17.m/3 **Reason:** {*AI05-0067-1*} This is necessary to prevent elaboration problems with deferred constants of controlled types. Consider:

```
package P is                                                                    17.m.1/3
   type Dyn_String is private;
   Null_String : constant Dyn_String;
   ...
private
   type Dyn_String is new Ada.Finalization.Controlled with ...
   procedure Finalize(X : in out Dyn_String);
   procedure Adjust(X : in out Dyn_String);

   Null_String : constant Dyn_String :=
      (Ada.Finalization.Controlled with ...);
   ...
end P;
```

When Null_String is elaborated, the bodies of Finalize and Adjust clearly have not been elaborated. Without this rule, this declaration would necessarily raise Program_Error (unless the permissions given below are used by the implementation). 17.m.2/3

Ramification: An aggregate with a controlled part used in the return expression of a simple_return_statement has to be built in place in the anonymous return object, as this is similar to an object declaration. (This is a change from Ada 95, but it is not an inconsistency as it only serves to restrict implementation choices.) But this only covers the aggregate; a separate anonymous return object can still be used unless it too is required to be built in place. 17.n/3

Similarly, an aggregate that has a controlled part but is not itself controlled and that is used to initialize an object also has to be built in place. This is also a change from Ada 95, but it is not an inconsistency as it only serves to restrict implementation choices. This avoids problems if a type like Dyn_String (in the example above) is used as a component in a type used as a deferred constant in package P. 17.o/3

- In other cases, it is unspecified whether the anonymous object is built in place. 17.4/3

Reason: This is left unspecified so the implementation can use any appropriate criteria for determining when to build in place. That includes making the decision on a call-by-call basis. Reasonable programs will not care what decision is made here anyway. 17.p/3

{*AI05-0067-1*} Notwithstanding what this document says elsewhere, if an object is built in place: 17.5/3

- Upon successful completion of the return statement or aggregate, the anonymous object *mutates into* the newly-created object; that is, the anonymous object ceases to exist, and the newly-created object appears in its place. 17.6/3
- Finalization is not performed on the anonymous object. 17.7/3
- Adjustment is not performed on the newly-created object. 17.8/3
- All access values that designate parts of the anonymous object now designate the corresponding parts of the newly-created object. 17.9/3
- All renamings of parts of the anonymous object now denote views of the corresponding parts of the newly-created object. 17.10/3
- Coextensions of the anonymous object become coextensions of the newly-created object. 17.11/3

To be honest: {*AI12-0005-1*} This "mutating" does not necessarily happen atomically with respect to abort and other tasks. For example, if a function call is used as the ancestor part of an extension_aggregate, then the tag of the anonymous object (the function result) will be different from the tag of the newly-created object (the ancestor part of the extension_aggregate). In implementation terms, this involves modifying the tag field. If the current task is aborted during this modification, the object might become abnormal. Likewise, if some other task accesses the tag field during this modification, it constitutes improper use of shared variables, and is erroneous. 17.q/5

Implementation Note: The intended implementation is that the anonymous object is allocated at the same address as the newly-created object. Thus, no run-time action is required to cause all the access values and renamings to point to the right place. They just point to the newly-created object, which is what the return object has magically "mutated into". 17.r/3

There is no requirement that 'Address of the return object is equal to 'Address of the newly-created object, but that will be true in the intended implementation. 17.s/3

For a function call, if the size of the newly-created object is known at the call site, the object is allocated there, and the address is implicitly passed to the function; the return object is created at that address. Otherwise, a storage pool is implicitly passed to the function; the size is determined at the point of the return statement, and passed to the Allocate procedure. The address returned by the storage pool is returned from the function, and the newly-created object uses 17.t/3

that same address. If the return statement is left without returning (via an exception or a goto, for example), then Deallocate is called. The storage pool might be a dummy pool that represents "allocate on the stack".

17.u/3 The Tag of the newly-created object may be different from that of the result object. Likewise, the master and accessibility level may be different.

17.v/3 An alternative implementation model might allow objects to move around to different addresses. In this case, access values and renamings would need to be modified at run time. It seems that this model requires the full power of tracing garbage collection.

Implementation Permissions

18/3 {*AI05-0067-1*} An implementation is allowed to relax the above rules for assignment_statements in the following ways:

18.a/3 *This paragraph was deleted.*{*AI05-0067-1*}

18.b/5 **Ramification:** {*AI05-0067-1*} {*AI12-0005-1*} The relaxations apply only to nonlimited types, as assignment_statements are not allowed for limited types. This is important so that the programmer can count on stricter semantics for limited controlled types.

19/5 • {*AI05-0067-1*} {*AI12-0444-1*} If an object is assigned the value of that same object, the implementation may omit the entire assignment.

19.a **Ramification:** In other words, even if an object is controlled and a combination of Finalize and Adjust on the object might have a net side effect, they need not be performed.

20/3 • {*AI05-0067-1*} For assignment of a noncontrolled type, the implementation may finalize and assign each component of the variable separately (rather than finalizing the entire variable and assigning the entire new value) unless a discriminant of the variable is changed by the assignment.

20.a **Reason:** For example, in a slice assignment, an anonymous object is not necessary if the slice is copied component-by-component in the right direction, since array types are not controlled (although their components may be). Note that the direction, and even the fact that it's a slice assignment, can in general be determined only at run time.

20.b/3 **Ramification:** {*AI05-0005-1*} This potentially breaks a single assignment operation into many, and thus abort deferral (see 9.8) needs to last only across an individual component assignment when the component has a controlled part. It is only important that the copy step is not separated (by an abort) from the adjust step, so aborts between component assignments is not harmful.

21/5 • {*AI95-00147-01*} {*AI05-0067-1*} {*AI12-0444-1*} The implementation may avoid creating an anonymous object if the value being assigned is the result of evaluating a name denoting an object (the source object) whose storage cannot overlap with the target. If the source object can overlap with the target object, then the implementation can avoid the need for an intermediary anonymous object by exercising one of the above permissions and perform the assignment one component at a time (for an overlapping array assignment), or not at all (for an assignment where the target and the source of the assignment are the same object).

21.a/3 **Ramification:** {*AI05-0005-1*} If the anonymous object is eliminated by this permission, there is no anonymous object to be finalized and thus the Finalize call on it is eliminated.

21.b/3 {*AI95-00147-01*} {*AI05-0005-1*} Note that if the anonymous object is eliminated but the new value is not built in place in the target object, that Adjust must be called directly on the target object as the last step of the assignment, since some of the subcomponents may be self-referential or otherwise position-dependent. This Adjust can be eliminated only by using one of the following permissions.

22/2 {*AI95-00147-01*} Furthermore, an implementation is permitted to omit implicit Initialize, Adjust, and Finalize calls and associated assignment operations on an object of a nonlimited controlled type provided that:

23/2 • any omitted Initialize call is not a call on a user-defined Initialize procedure, and

23.a/2 **To be honest:** This does not apply to any calls to a user-defined Initialize routine that happen to occur in an Adjust or Finalize routine. It is intended that it is never necessary to look inside of an Adjust or Finalize routine to determine if the call can be omitted.

23.b/2 **Reason:** We don't want to eliminate objects for which the Initialize might have side effects (such as locking a resource).

- any usage of the value of the object after the implicit Initialize or Adjust call and before any subsequent Finalize call on the object does not change the external effect of the program, and 24/2
- after the omission of such calls and operations, any execution of the program that executes an Initialize or Adjust call on an object or initializes an object by an aggregate will also later execute a Finalize call on the object and will always do so prior to assigning a new value to the object, and 25/2
- the assignment operations associated with omitted Adjust calls are also omitted. 26/2

This permission applies to Adjust and Finalize calls even if the implicit calls have additional external effects. 27/2

Reason: The goal of the above permissions is to allow typical dead assignment and dead variable removal algorithms to work for nonlimited controlled types. We require that "pairs" of Initialize/Adjust/Finalize operations are removed. (These aren't always pairs, which is why we talk about "any execution of the program".) 27.a/2

Extensions to Ada 83

Controlled types and user-defined finalization are new to Ada 95. (Ada 83 had finalization semantics only for masters of tasks.) 27.b

Extensions to Ada 95

{*AI95-00161-01*} **Amendment Correction:** Types Controlled and Limited_Controlled now have Preelaborable_Initialization, so that objects of types derived from these types can be used in preelaborated packages. 27.c/2

Wording Changes from Ada 95

{*8652/0020*} {*AI95-00126-01*} **Corrigendum:** Clarified that Ada.Finalization is a remote types package. 27.d/2

{*8652/0021*} {*AI95-00182-01*} **Corrigendum:** Added wording to clarify that the default initialization (whatever it is) of an ancestor part is used. 27.e/2

{*8652/0022*} {*AI95-00083-01*} **Corrigendum:** Clarified that Adjust is never called on an aggregate used for the initialization of an object or subaggregate, or passed as a parameter. 27.f/2

{*AI95-00147-01*} Additional optimizations are allowed for nonlimited controlled types. These allow traditional dead variable elimination to be applied to such types. 27.g/2

{*AI95-00318-02*} Corrected the build-in-place requirement for controlled aggregates to be consistent with the requirements for limited types. 27.h/2

{*AI95-00348-01*} The operations of types Controlled and Limited_Controlled are now declared as null procedures (see 6.7) to make the semantics clear (and to provide a good example of what null procedures can be used for). 27.i/2

{*AI95-00360-01*} Types that need finalization are defined; this is used by the No_Nested_Finalization restriction (see D.7, "Tasking Restrictions"). 27.j/2

{*AI95-00373-01*} Generalized the description of objects that have Initialize called for them to say that it is done for all objects that are initialized by default. This is needed so that all of the new cases are covered. 27.k/2

Extensions to Ada 2005

{*AI05-0212-1*} Package Ada.Finalization now has Pure categorization, so it can be mentioned for any package. Note that this does not change the preelaborability of objects descended from Controlled and Limited_Controlled. 27.l/3

Wording Changes from Ada 2005

{*AI05-0013-1*} **Correction:** Eliminated coextensions from the "needs finalization" rules, as this cannot be determined in general in the compilation unit that declares the type. (The designated type of the coextension may have been imported as a limited view.) Uses of "needs finalization" need to ensure that coextensions are handled by other means (such as in No_Nested_Finalization – see D.7) or that coextensions cannot happen. 27.m/3

{*AI05-0013-1*} **Correction:** Corrected the "needs finalization" rules to include class-wide types, as a future extension can include a part that needs finalization. 27.n/3

{*AI05-0026-1*} **Correction:** Corrected the "needs finalization" rules to clearly say that they ignore privacy. 27.o/3

{*AI05-0067-1*} **Correction:** Changed "built in place" to Dynamic Semantics and centralized the rules here. This eliminates the fiction that built in place is just a combination of a permission and a requirement; it clearly has noticeable semantic effects. This wording change is not intended to change the semantics of any correct Ada program. 27.p/3

7.6.1 Completion and Finalization

1 [This subclause defines *completion* and *leaving* of the execution of constructs and entities. A *master* is the execution of a construct that includes finalization of local objects after it is complete (and after waiting for any local tasks — see 9.3), but before leaving. Other constructs and entities are left immediately upon completion.]

Dynamic Semantics

2/2 {*AI95-00318-02*} The execution of a construct or entity is *complete* when the end of that execution has been reached, or when a transfer of control (see 5.1) causes it to be abandoned. Completion due to reaching the end of execution, or due to the transfer of control of an exit_statement, return statement, goto_statement, or requeue_statement or of the selection of a terminate_alternative is *normal completion*. Completion is *abnormal* otherwise [— when control is transferred out of a construct due to abort or the raising of an exception].

2.a **Discussion:** Don't confuse the run-time concept of completion with the compile-time concept of completion defined in 3.11.1.

3/5 {*AI95-00162-01*} {*AI95-00416-01*} {*AI12-0406-1*} After execution of a construct or entity is complete, it is *left*, meaning that execution continues with the next action, as defined for the execution that is taking place. Leaving an execution happens immediately after its completion, except in the case of the execution of a *master construct*: a body other than a package_body; a statement; or an expression, function_call, or range that is not part of an enclosing expression, function_call, range, or simple_statement other than a simple_return_statement. The term *master* by itself refers to the execution of a master construct. A master is finalized after it is complete, and before it is left.

3.a/2 **Reason:** {*AI95-00162-01*} {*AI95-00416-01*} Expressions and statements are masters so that objects created by subprogram calls (in aggregates, allocators for anonymous access-to-object types, and so on) are finalized and have their tasks awaited before the expressions or statements are left. Note that expressions like the condition of an if_statement are masters, because they are not enclosed by a simple_statement. Similarly, a function_call which is renamed is a master, as it is not in a simple_statement.

3.b/2 {*AI95-00416-01*} We have to include function_calls in the contexts that do not cause masters to occur so that expressions contained in a function_call (that is not part of an expression or simple_statement) do not individually become masters. We certainly do not want the parameter expressions of a function_call to be separate masters, as they would then be finalized before the function is called.

3.c/2 **Ramification:** {*AI95-00416-01*} The fact that a function_call is a master does not change the accessibility of the return object denoted by the function_call; that depends on the use of the function_call. The function_call is the master of any short-lived entities (such as aggregates used as parameters of types with task or controlled parts).

3.d/5 **Term entry: master** — execution of a master construct
Note: Each object and task is associated with a master. When a master is left, associated tasks are awaited and associated objects are finalized.

3.e/5 **Term entry: master construct** — one of certain executable constructs for which there can be objects or tasks whose lifetime ends when the construct completes
Note: Execution of a master construct is a master, with which objects and tasks are associated for the purposes of waiting and finalization.

4 For the *finalization* of a master, dependent tasks are first awaited, as explained in 9.3. Then each object whose accessibility level is the same as that of the master is finalized if the object was successfully initialized and still exists. [These actions are performed whether the master is left by reaching the last statement or via a transfer of control.] When a transfer of control causes completion of an execution, each included master is finalized in order, from innermost outward.

4.a **Ramification:** As explained in 3.10.2, the set of objects with the same accessibility level as that of the master includes objects declared immediately within the master, objects declared in nested packages, objects created by allocators (if the ultimate ancestor access type is declared in one of those places) and subcomponents of all of these things. If an object was already finalized by Unchecked_Deallocation, then it is not finalized again when the master is left.

Note that any object whose accessibility level is deeper than that of the master would no longer exist; those objects would have been finalized by some inner master. Thus, after leaving a master, the only objects yet to be finalized are those whose accessibility level is less deep than that of the master. 4.b

To be honest: Subcomponents of objects due to be finalized are not finalized by the finalization of the master; they are finalized by the finalization of the containing object. 4.c

Reason: We need to finalize subcomponents of objects even if the containing object is not going to get finalized because it was not fully initialized. But if the containing object is finalized, we don't want to require repeated finalization of the subcomponents, as might normally be implied by the recursion in finalization of a master and the recursion in finalization of an object. 4.d

To be honest: Formally, completion and leaving refer to executions of constructs or entities. However, the standard sometimes (informally) refers to the constructs or entities whose executions are being completed. Thus, for example, "the subprogram call or task is complete" really means "*the execution of* the subprogram call or task is complete". 4.e

For the *finalization* of an object: 5

- {*AI05-0099-1*} If the full type of the object is an elementary type, finalization has no effect; 6/3

  Reason: {*AI05-0099-1*} We say "full type" in this and the following bullets as privacy is ignored for the purpose of determining the finalization actions of an object; that is as expected for Dynamic Semantics rules. 6.a/3

- {*AI05-0099-1*} If the full type of the object is a tagged type, and the tag of the object identifies a controlled type, the Finalize procedure of that controlled type is called; 7/3

- {*AI05-0099-1*} If the full type of the object is a protected type, or if the full type of the object is a tagged type and the tag of the object identifies a protected type, the actions defined in 9.4 are performed; 8/3

- {*AI95-00416-01*} {*AI05-0099-1*} If the full type of the object is a composite type, then after performing the above actions, if any, every component of the object is finalized in an arbitrary order, except as follows: if the object has a component with an access discriminant constrained by a per-object expression, this component is finalized before any components that do not have such discriminants; for an object with several components with such a discriminant, they are finalized in the reverse of the order of their component_declarations; 9/3

  Reason: This allows the finalization of a component with an access discriminant to refer to other components of the enclosing object prior to their being finalized. 9.a

  To be honest: {*AI05-0099-1*} {*AI12-0005-1*} The components discussed here are all of the components that the object actually has, not just those components that are statically identified by the type of the object. These can be different if the object has a class-wide type. 9.b/4

- {*AI95-00416-01*} If the object has coextensions (see 3.10.2), each coextension is finalized after the object whose access discriminant designates it. 9.1/2

  Ramification: {*AI05-0066-1*} In the case of an aggregate or function call that is used (in its entirety) to directly initialize a part of an object, the coextensions of the result of evaluating the aggregate or function call are transfered to become coextensions of the object being initialized and are not finalized until the object being initialized is ultimately finalized, even if an anonymous object is created as part of the operation. 9.c/3

Immediately before an instance of Unchecked_Deallocation reclaims the storage of an object, the object is finalized. [If an instance of Unchecked_Deallocation is never applied to an object created by an allocator, the object will still exist when the corresponding master completes, and it will be finalized then.] 10

{*AI95-00280-01*} {*AI05-0051-1*} {*AI05-0190-1*} The finalization of a master performs finalization of objects created by declarations in the master in the reverse order of their creation. After the finalization of a master is complete, the objects finalized as part of its finalization cease to *exist*, as do any types and subtypes defined and created within the master. 11/3

This paragraph was deleted.{*AI05-0190-1*} 11.a/3

This paragraph was deleted.{*AI05-0190-1*} 11.b/3

This paragraph was deleted.{*AI05-0190-1*} 11.c/3

This paragraph was deleted.{*AI05-0190-1*} 11.d/3

11.e **Ramification:** Note that a deferred constant declaration does not create the constant; the full constant declaration creates it. Therefore, the order of finalization depends on where the full constant declaration occurs, not the deferred constant declaration.

11.f An imported object is not created by its declaration. It is neither initialized nor finalized.

11.g **Implementation Note:** An implementation has to ensure that the storage for an object is not reclaimed when references to the object are still possible (unless, of course, the user explicitly requests reclamation via an instance of Unchecked_Deallocation). This implies, in general, that objects cannot be deallocated one by one as they are finalized; a subsequent finalization might reference an object that has been finalized, and that object had better be in its (well-defined) finalized state.

11.1/3 {*AI05-0190-1*} Each nonderived access type *T* has an associated *collection*, which is the set of objects created by allocators of *T*, or of types derived from *T*. Unchecked_Deallocation removes an object from its collection. Finalization of a collection consists of finalization of each object in the collection, in an arbitrary order. The collection of an access type is an object implicitly declared at the following place:

11.h/3 **Ramification:** {*AI05-0190-1*} The place of the implicit declaration determines when allocated objects are finalized. For multiple collections declared at the same place, we do not define the order of their implicit declarations.

11.i/3 {*AI05-0190-1*} Finalization of allocated objects is done according to the (ultimate ancestor) allocator type, not according to the storage pool in which they are allocated. Pool finalization might reclaim storage (see 13.11, "Storage Management"), but has nothing (directly) to do with finalization of the pool elements.

11.j/3 {*AI05-0190-1*} Note that finalization is done only for objects that still exist; if an instance of Unchecked_Deallocation has already gotten rid of a given pool element, that pool element will not be finalized when the master is left.

11.k/3 **Reason:** {*AI05-0190-1*} Note that we talk about the type of the allocator here. There may be access values of a (general) access type pointing at objects created by allocators for some other type; these are not (necessarily) finalized at this point.

11.2/3 - For a named access type, the first freezing point (see 13.14) of the type.

11.l/3 **Reason:** {*AI05-0190-1*} The freezing point of the ultimate ancestor access type is chosen because before that point, pool elements cannot be created, and after that point, access values designating (parts of) the pool elements can be created. This is also the point after which the pool object cannot have been declared. We don't want to finalize the pool elements until after anything finalizing objects that contain access values designating them. Nor do we want to finalize pool elements after finalizing the pool object itself.

11.3/3 - For the type of an access parameter, the call that contains the allocator.

11.4/3 - For the type of an access result, within the master of the call (see 3.10.2).

11.m/3 **To be honest:** {*AI05-0005-1*} {*AI05-0190-1*} We mean at a place within the master consistent with the execution of the call within the master. We don't say that normatively, as it is difficult to explain that when the master of the call need not be the master that immediately includes the call (such as when an anonymous result is converted to a named access type).

11.5/3 - For any other anonymous access type, the first freezing point of the innermost enclosing declaration.

12/2 {*AI95-00256-01*} The target of an assignment_statement is finalized before copying in the new value, as explained in 7.6.

13/3 {*8652/0021*} {*AI95-00182-01*} {*AI95-00162-01*} {*AI05-0066-1*} {*AI05-0142-4*} {*AI05-0269-1*} The master of an object is the master enclosing its creation whose accessibility level (see 3.10.2) is equal to that of the object, except in the case of an anonymous object representing the result of an aggregate or function call. If such an anonymous object is part of the result of evaluating the actual parameter expression for an explicitly aliased parameter of a function call, the master of the object is the innermost master enclosing the evaluation of the aggregate or function call, excluding the aggregate or function call itself. Otherwise, the master of such an anonymous object is the innermost master enclosing the evaluation of the aggregate or function call, which may be the aggregate or function call itself.

13.a/2 *This paragraph was deleted.*{*AI95-00162-01*}

13.b/2 `This paragraph was deleted.`

13.c/2 *This paragraph was deleted.*

Reason: {*AI95-00162-01*} This effectively imports all of the special rules for the accessibility level of renames, allocators, and so on, and applies them to determine where objects created in them are finalized. For instance, the master of a rename of a subprogram is that of the renamed subprogram. 13.d/2

{*AI05-0066-1*} In 3.10.2 we assign an accessibility level to the result of an aggregate or function call that is used to directly initialize a part of an object based on the object being initialized. This is important to ensure that any access discriminants denote objects that live at least as long as the object being initialized. However, if the result of the aggregate or function call is not built directly in the target object, but instead is built in an anonymous object that is then assigned to the target, the anonymous object needs to be finalized after the assignment rather than persisting until the target object is finalized (but not its coextensions). (Note than an implementation is never required to create such an anonymous object, and in some cases is required to *not* have such a separate object, but rather to build the result directly in the target.) 13.e/3

{*AI05-0142-4*} The special case for explicitly aliased parameters of functions is needed for the same reason, as access discriminants of the returned object may designate one of these parameters. In that case, we want to lengthen the lifetime of the anonymous objects as long as the possible lifetime of the result. 13.f/3

{*AI05-0142-4*} We don't do a similar change for other kinds of calls, because the extended lifetime of the parameters adds no value, but could constitute a storage leak. For instance, such an anonymous object created by a procedure call in the elaboration part of a package body would have to live until the end of the program, even though it could not be used after the procedure returns (other than via Unchecked_Access). 13.g/3

Ramification: {*AI05-0142-4*} Note that the lifetime of the master given to anonymous objects in explicitly aliased parameters of functions is not necessarily as long as the lifetime of the master of the object being initialized (if the function call is used to initialize an allocator, for instance). In that case, the accessibility check on explicitly aliased parameters will necessarily fail if any such anonymous objects exist. This is necessary to avoid requiring the objects to live as long as the access type or having the implementation complexity of an implicit coextension. 13.h/3

{*8652/0023*} {*AI95-00169-01*} {*AI95-00162-01*} {*AI05-0066-1*} {*AI05-0262-1*} In the case of an expression that is a master, finalization of any (anonymous) objects occurs after completing evaluation of the expression and all use of the objects, prior to starting the execution of any subsequent construct. 13.1/3

Bounded (Run-Time) Errors

{*8652/0023*} {*AI95-00169-01*} It is a bounded error for a call on Finalize or Adjust that occurs as part of object finalization or assignment to propagate an exception. The possible consequences depend on what action invoked the Finalize or Adjust operation: 14/1

Ramification: It is not a bounded error for Initialize to propagate an exception. If Initialize propagates an exception, then no further calls on Initialize are performed, and those components that have already been initialized (either explicitly or by default) are finalized in the usual way. 14.a

{*8652/0023*} {*AI95-00169-01*} It also is not a bounded error for an explicit call to Finalize or Adjust to propagate an exception. We do not want implementations to have to treat explicit calls to these routines specially. 14.a.1/1

- For a Finalize invoked as part of an assignment_statement, Program_Error is raised at that point. 15
- {*8652/0024*} {*AI95-00193-01*} {*AI95-00256-01*} {*AI12-0445-1*} For an Adjust invoked as part of assignment operations other than those invoked as part of an assignment_statement, some of the adjustments due to be performed can be performed, and then Program_Error is raised. During its propagation, finalization may be applied to objects whose Adjust failed. For an Adjust invoked as part of an assignment_statement, any other adjustments due to be performed are performed, and then Program_Error is raised. 16/5

Reason: {*8652/0024*} {*AI95-00193-01*} {*AI95-00256-01*} In the case of assignments that are part of initialization, there is no need to complete all adjustments if one propagates an exception, as the object will immediately be finalized. So long as a subcomponent is not going to be finalized, it need not be adjusted, even if it is initialized as part of an enclosing composite assignment operation for which some adjustments are performed. However, there is no harm in an implementation making additional Adjust calls (as long as any additional components that are adjusted are also finalized), so we allow the implementation flexibility here. On the other hand, for an assignment_statement, it is important that all adjustments be performed, even if one fails, because all controlled subcomponents are going to be finalized. Other kinds of assignment are more like initialization than assignment_statements, so we include them as well in the permission. 16.a/2

Ramification: {*8652/0024*} {*AI95-00193-01*} Even if an Adjust invoked as part of the initialization of a controlled object propagates an exception, objects whose initialization (including any Adjust or Initialize calls) successfully 16.a.1/1

completed will be finalized. The permission above only applies to objects whose Adjust failed. Objects for which Adjust was never even invoked must not be finalized.

17 • For a Finalize invoked as part of a call on an instance of Unchecked_Deallocation, any other finalizations due to be performed are performed, and then Program_Error is raised.

17.a.1/1 **Discussion:** {*8652/0104*} {*AI95-00179-01*} The standard does not specify if storage is recovered in this case. If storage is not recovered (and the object continues to exist), Finalize may be called on the object again (when the allocator's master is finalized).

17.1/3 • *This paragraph was deleted.*{*8652/0023*} {*AI95-00169-01*} {*AI05-0064-1*}

17.2/1 • {*8652/0023*} {*AI95-00169-01*} For a Finalize invoked due to reaching the end of the execution of a master, any other finalizations associated with the master are performed, and Program_Error is raised immediately after leaving the master.

17.a/3 **Discussion:** {*AI05-0064-1*} This rule covers both ordinary objects created by a declaration, and anonymous objects created as part of evaluating an expression. All contexts that create objects that need finalization are defined to be masters.

18/2 • {*AI95-00318-02*} For a Finalize invoked by the transfer of control of an exit_statement, return statement, goto_statement, or requeue_statement, Program_Error is raised no earlier than after the finalization of the master being finalized when the exception occurred, and no later than the point where normal execution would have continued. Any other finalizations due to be performed up to that point are performed before raising Program_Error.

18.a **Ramification:** For example, upon leaving a block_statement due to a goto_statement, the Program_Error would be raised at the point of the target statement denoted by the label, or else in some more dynamically nested place, but not so nested as to allow an exception_handler that has visibility upon the finalized object to handle it. For example,

18.b
```
procedure Main is
begin
    <<The_Label>>
    Outer_Block_Statement : declare
        X : Some_Controlled_Type;
    begin
        Inner_Block_Statement : declare
            Y : Some_Controlled_Type;
            Z : Some_Controlled_Type;
        begin
            goto The_Label;
        exception
            when Program_Error => ... -- Handler number 1.
        end;
    exception
        when Program_Error => ... -- Handler number 2.
    end;
exception
    when Program_Error => ... -- Handler number 3.
end Main;
```

18.c/5 {*AI12-0005-1*} The goto_statement will first cause Finalize(Z) to be called. Suppose that Finalize(Z) propagates an exception. Program_Error will be raised after leaving Inner_Block_Statement, but before leaving Main. Thus, handler number 1 cannot handle this Program_Error; it will be handled either by handler number 2 or handler number 3. If it is handled by handler number 2, then Finalize(Y) will be done before executing the handler. If it is handled by handler number 3, then Finalize(Y) and Finalize(X) will both be done before executing the handler.

19 • For a Finalize invoked by a transfer of control that is due to raising an exception, any other finalizations due to be performed for the same master are performed; Program_Error is raised immediately after leaving the master.

19.a/5 **Ramification:** {*AI12-0005-1*} If, in the above example, the goto_statement were replaced by a raise_statement, then the Program_Error would be handled by handler number 2, and Finalize(Y) would be done before executing the handler.

19.b **Reason:** We considered treating this case in the same way as the others, but that would render certain exception_handlers useless. For example, suppose the only exception_handler is one for **others** in the main subprogram. If some deeply nested call raises an exception, causing some Finalize operation to be called, which then raises an exception, then normal execution "would have continued" at the beginning of the exception_handler. Raising Program_Error at that point would cause that handler's code to be skipped. One would need two nested exception_handlers to be sure of catching such cases!

On the other hand, the exception_handler for a given master should not be allowed to handle exceptions raised during finalization of that master. 19.c

- For a Finalize invoked by a transfer of control due to an abort or selection of a terminate alternative, the exception is ignored; any other finalizations due to be performed are performed. 20

Ramification: This case includes an asynchronous transfer of control. 20.a

To be honest: This violates the general principle that it is always possible for a bounded error to raise Program_Error (see 1.1.5, “Classification of Errors”). 20.b

Implementation Permissions

{*AI05-0107-1*} If the execution of an allocator propagates an exception, any parts of the allocated object that were successfully initialized may be finalized as part of the finalization of the innermost master enclosing the allocator. 20.1/3

Reason: This allows deallocating the memory for the allocated object at the innermost master, preventing a storage leak. Otherwise, the object would have to stay around until the finalization of the collection that it belongs to, which could be the entire life of the program if the associated access type is library level. 20.c/3

{*AI05-0111-3*} {*AI05-0262-1*} The implementation may finalize objects created by allocators for an access type whose storage pool supports subpools (see 13.11.4) as if the objects were created (in an arbitrary order) at the point where the storage pool was elaborated instead of at the first freezing point of the access type. 20.2/3

Ramification: This allows the finalization of such objects to occur later than they otherwise would, but still as part of the finalization of the same master. Accessibility rules in 13.11.4 ensure that it is the same master (usually that of the environment task). 20.d/3

Implementation Note: {*AI12-0005-1*} This permission is intended to allow the allocated objects to "belong" to the subpool objects and to allow those objects to be finalized at the time that the storage pool is finalized (if they are not finalized earlier). This is expected to ease implementation, as the remaining yet-to-be deallocated objects will only need to be accessible at run time from the subpool header and not also from the overall access type collection header. That is, they only need to belong to a single list, rather than two. 20.e/5

NOTE 1 {*AI05-0299-1*} {*AI12-0449-1*} The rules of Clause 10 imply that immediately prior to partition termination, Finalize operations are applied to library-level controlled objects (including those created by allocators of library-level access types, except those already finalized). This occurs after waiting for library-level tasks to terminate. 21/5

Discussion: We considered defining a pragma that would apply to a controlled type that would suppress Finalize operations for library-level objects of the type upon partition termination. This would be useful for types whose finalization actions consist of simply reclaiming global heap storage, when this is already provided automatically by the environment upon program termination. 21.a

NOTE 2 A constant is only constant between its initialization and finalization. Both initialization and finalization are allowed to change the value of a constant. 22

NOTE 3 Abort is deferred during certain operations related to controlled types, as explained in 9.8. Those rules prevent an abort from causing a controlled object to be left in an ill-defined state. 23

NOTE 4 The Finalize procedure is called upon finalization of a controlled object, even if Finalize was called earlier, either explicitly or as part of an assignment; hence, if a controlled type is visibly controlled (implying that its Finalize primitive is directly callable), or is nonlimited (implying that assignment is allowed), its Finalize procedure is ideally designed to have no ill effect if it is applied a second time to the same object. 24

Discussion: Or equivalently, a Finalize procedure should be “idempotent”; applying it twice to the same object should be equivalent to applying it once. 24.a

Reason: A user-written Finalize procedure should be idempotent since it can be called explicitly by a client (at least if the type is "visibly" controlled). Also, Finalize is used implicitly as part of the assignment_statement if the type is nonlimited, and an abort is permitted to disrupt an assignment_statement between finalizing the left-hand side and assigning the new value to it (an abort is not permitted to disrupt an assignment operation between copying in the new value and adjusting it). 24.b

Discussion: {*AI95-00287-01*} Either Initialize or Adjust, but not both, is applied to (almost) every controlled object when it is created: Initialize is done when no initial value is assigned to the object, whereas Adjust is done as part of assigning the initial value. The one exception is the object initialized by an aggregate (both the anonymous object created for an aggregate, or an object initialized by an aggregate that is built-in-place); Initialize is not applied to the aggregate as a whole, nor is the value of the aggregate or object adjusted. 24.c/2

24.d All of the following use the assignment operation, and thus perform value adjustment:

24.e
- the assignment_statement (see 5.2);

24.f
- explicit initialization of a stand-alone object (see 3.3.1) or of a pool element (see 4.8);

24.g
- default initialization of a component of a stand-alone object or pool element (in this case, the value of each component is assigned, and therefore adjusted, but the value of the object as a whole is not adjusted);

24.h/2
- {*AI95-00318-02*} function return, when the result is not built-in-place (adjustment of the result happens before finalization of the function);

24.i
- predefined operators (although the only one that matters is concatenation; see 4.5.3);

24.j
- generic formal objects of mode **in** (see 12.4); these are defined in terms of constant declarations; and

24.k/5
- {*AI95-00287-01*} {*AI12-0005-1*} aggregates (see 4.3), when the result is not built-in-place (in this case, the value of each component, and the ancestor part, for an extension_aggregate, is assigned, and therefore adjusted, but the value of the aggregate as a whole is not adjusted; neither is Initialize called);

24.l The following also use the assignment operation, but adjustment never does anything interesting in these cases:

24.m
- By-copy parameter passing uses the assignment operation (see 6.4.1), but controlled objects are always passed by reference, so the assignment operation never does anything interesting in this case. If we were to allow by-copy parameter passing for controlled objects, we would need to make sure that the actual is finalized before doing the copy back for [**in**] **out** parameters. The finalization of the parameter itself needs to happen after the copy back (if any), similar to the finalization of an anonymous function return object or aggregate object.

24.n
- **For** loops use the assignment operation (see 5.5), but since the type of the loop parameter is never controlled, nothing interesting happens there, either.

24.n.1/2
- {*AI95-00318-02*} Objects initialized by function results and aggregates that are built-in-place. In this case, the assignment operation is never executed, and no adjustment takes place. While built-in-place is always allowed, it is required for some types — see 7.5 and 7.6 — and that's important since limited types have no Adjust to call.

24.o/2 *This paragraph was deleted.*{*AI95-00287-01*}

24.p Finalization of the parts of a protected object are not done as protected actions. It is possible (in pathological cases) to create tasks during finalization that access these parts in parallel with the finalization itself. This is an erroneous use of shared variables.

24.q **Implementation Note:** One implementation technique for finalization is to chain the controlled objects together on a per-task list. When leaving a master, the list can be walked up to a marked place. The links needed to implement the list can be declared (privately) in types Controlled and Limited_Controlled, so they will be inherited by all controlled types.

24.r Another implementation technique, which we refer to as the "PC-map" approach essentially implies inserting exception handlers at various places, and finalizing objects based on where the exception was raised.

24.s The PC-map approach is for the compiler/linker to create a map of code addresses; when an exception is raised, or abort occurs, the map can be consulted to see where the task was executing, and what finalization needs to be performed. This approach was given in the Ada 83 Rationale as a possible implementation strategy for exception handling — the map is consulted to determine which exception handler applies.

24.t If the PC-map approach is used, the implementation must take care in the case of arrays. The generated code will generally contain a loop to initialize an array. If an exception is raised part way through the array, the components that have been initialized must be finalized, and the others must not be finalized.

24.u It is our intention that both of these implementation methods should be possible.

Wording Changes from Ada 83

24.v Finalization depends on the concepts of completion and leaving, and on the concept of a master. Therefore, we have moved the definitions of these concepts here, from where they used to be in Clause 9. These concepts also needed to be generalized somewhat. Task waiting is closely related to user-defined finalization; the rules here refer to the task-waiting rules of Clause 9.

Inconsistencies With Ada 95

24.v.1/3 {*AI05-0066-1*} **Ada 2012 Correction:** Changed the definition of the master of an anonymous object used to directly initialize an object, so it can be finalized immediately rather than having to hang around as long as the object. In this case, the Ada 2005 definition was inconsistent with Ada 95, and Ada 2012 changes it back. It is unlikely that many compilers implemented the rule as written in Amendment 1, so an inconsistency is unlikely to arise in practice.

Wording Changes from Ada 95

{*8652/0021*} {*AI95-00182-01*} **Corrigendum:** Fixed the wording to say that anonymous objects aren't finalized until the object can't be used anymore. 24.w/2

{*8652/0023*} {*AI95-00169-01*} **Corrigendum:** Added wording to clarify what happens when Adjust or Finalize raises an exception; some cases had been omitted. 24.x/2

{*8652/0024*} {*AI95-00193-01*} {*AI95-00256-01*} **Corrigendum:** Stated that if Adjust raises an exception during initialization, nothing further is required. This is corrected in Ada 2005 to include all kinds of assignment other than assignment_statements. 24.y/2

{*AI95-00162-01*} {*AI95-00416-01*} Revised the definition of master to include expressions and statements, in order to cleanly define what happens for tasks and controlled objects created as part of a subprogram call. Having done that, all of the special wording to cover those cases is eliminated (at least until the Ada comments start rolling in). 24.z/2

{*AI95-00280-01*} We define *finalization of the collection* here, so as to be able to conveniently refer to it in other rules (especially in 4.8, “Allocators”). 24.aa/2

{*AI95-00416-01*} Clarified that a coextension is finalized at the same time as the outer object. (This was intended for Ada 95, but since the concept did not have a name, it was overlooked.) 24.bb/2

Inconsistencies With Ada 2005

{*AI05-0051-1*} {*AI05-0190-1*} **Correction:** Better defined when objects allocated from anonymous access types are finalized. This could be inconsistent if objects are finalized in a different order than in an Ada 2005 implementation and that order caused different program behavior; however programs that depend on the order of finalization within a single master are already fragile and hopefully are rare. 24.cc/3

Wording Changes from Ada 2005

{*AI05-0064-1*} **Correction:** Removed a redundant rule, which is now covered by the additional places where masters are defined. 24.dd/3

{*AI05-0099-1*} {*AI12-0005-1*} **Correction:** Clarified the finalization rules so that there is no doubt that privacy is ignored, and to ensure that objects of class-wide interface types are finalized based on their specific concrete type. 24.ee/4

{*AI05-0107-1*} **Correction:** Allowed premature finalization of parts of failed allocators. This could be an inconsistency, but the previous behavior is still allowed and there is no requirement that implementations take advantage of the permission. 24.ff/3

{*AI05-0111-3*} Added a permission to finalize an object allocated from a subpool later than usual. 24.gg/3

{*AI05-0142-4*} Added text to specially define the master of anonymous objects which are passed as explicitly aliased parameters (see 6.1) of functions. The model for these parameters is explained in detail in 6.4.1. 24.hh/3

Wording Changes from Ada 2012

{*AI12-0406-1*} **Correction:** Defined the term “master construct”, so as to put static accessibility rules on a firmer basis, including ensuring that those rules apply inside of generic bodies. 24.ii/5

8 Visibility Rules

{*AI05-0299-1*} [The rules defining the scope of declarations and the rules defining which identifiers, character_literals, and operator_symbols are visible at (or from) various places in the text of the program are described in this clause. The formulation of these rules uses the notion of a declarative region. 1/3

{*AI12-0439-1*} As explained in Clause 3, a declaration declares a view of an entity and associates a defining name with that view. The view comprises an identification of the viewed entity, and possibly additional properties. A usage name denotes a declaration. It also denotes the view declared by that declaration, and denotes the entity of that view. Thus, two different usage names can denote two different views of the same entity; in this case they denote the same entity.] 2/5

To be honest: In some cases, a usage name that denotes a declaration does not denote the view declared by that declaration, nor the entity of that view, but instead denotes a view of the current instance of the entity, and denotes the current instance of the entity. This sometimes happens when the usage name occurs inside the declarative region of the declaration. 2.a

Wording Changes from Ada 83

We no longer define the term "basic operation;" thus we no longer have to worry about the visibility of them. Since they were essentially always visible in Ada 83, this change has no effect. The reason for this change is that the definition in Ada 83 was confusing, and not quite correct, and we found it difficult to fix. For example, one wonders why an if_statement was not a basic operation of type Boolean. For another example, one wonders what it meant for a basic operation to be "inherent in" something. Finally, this fixes the problem addressed by AI83-00027/07. 2.b

8.1 Declarative Region

Static Semantics

For each of the following constructs, there is a portion of the program text called its *declarative region*, [within which nested declarations can occur]: 1

- any declaration, other than that of an enumeration type, that is not a completion [of a previous declaration]; 2
- {*AI12-0094-1*} an access_definition; 2.1/4
- {*AI12-0061-1*} an iterated_component_association; 2.2/5
- {*AI12-0308-1*} an iterated_element_association; 2.3/5
- {*AI05-0255-1*} {*AI12-0308-1*} a quantified_expression; 2.4/5
- {*AI12-0236-1*} a declare_expression; 2.5/5
- a block_statement; 3
- a loop_statement; 4
- *This paragraph was deleted.*{*AI05-0255-1*} {*AI12-0308-1*} 4.1/5
- {*AI95-00318-02*} an extended_return_statement; 4.2/3
- an accept_statement; 5
- an exception_handler. 6

The declarative region includes the text of the construct together with additional text determined [(recursively)], as follows: 7

- If a declaration is included, so is its completion, if any. 8

9 • If the declaration of a library unit [(including Standard — see 10.1.1)] is included, so are the declarations of any child units [(and their completions, by the previous rule)]. The child declarations occur after the declaration.

10 • If a body_stub is included, so is the corresponding subunit.

11 • If a type_declaration is included, then so is a corresponding record_representation_clause, if any.

11.a **Reason:** This is so that the component_declarations can be directly visible in the record_representation_clause.

12 The declarative region of a declaration is also called the *declarative region* of any view or entity declared by the declaration.

12.a **Reason:** The constructs that have declarative regions are the constructs that can have declarations nested inside them. Nested declarations are declared in that declarative region. The one exception is for enumeration literals; although they are nested inside an enumeration type declaration, they behave as if they were declared at the same level as the type.

12.b **To be honest:** A declarative region does not include parent_unit_names.

12.c **Ramification:** A declarative region does not include context_clauses.

13 A declaration occurs *immediately within* a declarative region if this region is the innermost declarative region that encloses the declaration (the *immediately enclosing* declarative region), not counting the declarative region (if any) associated with the declaration itself.

13.a **Discussion:** Don't confuse the declarative region of a declaration with the declarative region in which it immediately occurs.

14 [A declaration is *local* to a declarative region if the declaration occurs immediately within the declarative region.] [An entity is *local* to a declarative region if the entity is declared by a declaration that is local to the declarative region.]

14.a **Ramification:** "Occurs immediately within" and "local to" are synonyms (when referring to declarations).

14.b Thus, “local to” applies to both declarations and entities, whereas “occurs immediately within” only applies to declarations. We use this term only informally; for cases where precision is required, we use the term "occurs immediately within", since it is less likely to cause confusion.

15 A declaration is *global* to a declarative region if the declaration occurs immediately within another declarative region that encloses the declarative region. An entity is *global* to a declarative region if the entity is declared by a declaration that is global to the declarative region.

16 NOTE 1 The children of a parent library unit are inside the parent's declarative region, even though they do not occur inside the parent's declaration or body. This implies that one can use (for example) "P.Q" to refer to a child of P whose defining name is Q, and that after "**use** P;" Q can refer (directly) to that child.

17 NOTE 2 As explained above and in 10.1.1, “Compilation Units - Library Units”, all library units are descendants of Standard, and so are contained in the declarative region of Standard. They are *not* inside the declaration or body of Standard, but they *are* inside its declarative region.

18/5 NOTE 3 {*AI12-0442-1*} For a declarative region that comes in multiple parts, the text of the declarative region does not include any of the text that appears between the parts. Thus, when a portion of a declarative region is said to extend from one place to another in the declarative region, the portion does not contain any of the text that appears between the parts of the declarative region.

18.a **Discussion:** It is necessary for the things that have a declarative region to include anything that contains declarations (except for enumeration type declarations). This includes any declaration that has a profile (that is, subprogram_declaration, subprogram_body, entry_declaration, subprogram_renaming_declaration, formal_subprogram_declaration, access-to-subprogram type_declaration), anything that has a discriminant_part (that is, various kinds of type_declaration), anything that has a component_list (that is, record type_declaration and record extension type_declaration), and finally the declarations of task and protected units and packages.

Wording Changes from Ada 83

18.b It was necessary to extend Ada 83's definition of declarative region to take the following Ada 95 features into account:

18.c • Child library units.

- Derived types/type extensions — we need a declarative region for inherited components and also for new components. 18.d
- All the kinds of types that allow discriminants. 18.e
- Protected units. 18.f
- Entries that have bodies instead of accept statements. 18.g
- The choice_parameter_specification of an exception_handler. 18.h
- The formal parameters of access-to-subprogram types. 18.i
- Renamings-as-body. 18.j

Discriminated and access-to-subprogram type declarations need a declarative region. Enumeration type declarations cannot have one, because you don't have to say "Color.Red" to refer to the literal Red of Color. For other type declarations, it doesn't really matter whether or not there is an associated declarative region, so for simplicity, we give one to all types except enumeration types. 18.k

We now say that an accept_statement has its own declarative region, rather than being part of the declarative region of the entry_declaration, so that declarative regions are properly nested regions of text, so that it makes sense to talk about "inner declarative regions", and "...extends to the end of a declarative region". Inside an accept_statement, the name of one of the parameters denotes the parameter_specification of the accept_statement, not that of the entry_declaration. If the accept_statement is nested within a block_statement, these parameter_specifications can hide declarations of the block_statement. The semantics of such cases was unclear in RM83. 18.l

To be honest: Unfortunately, we have the same problem for the entry name itself — it should denote the accept_statement, but accept_statements are not declarations. They should be, and they should hide the entry from all visibility within themselves. 18.m

Note that we can't generalize this to entry_bodies, or other bodies, because the declarative_part of a body is not supposed to contain (explicit) homographs of things in the declaration. It works for accept_statements only because an accept_statement does not have a declarative_part. 18.n

To avoid confusion, we use the term “local to” only informally in Ada 95. Even RM83 used the term incorrectly (see, for example, RM83-12.3(13)). 18.o

In Ada 83, (root) library units were inside Standard; it was not clear whether the declaration or body of Standard was meant. In Ada 95, they are children of Standard, and so occur immediately within Standard's declarative region, but not within either the declaration or the body. (See RM83-8.6(2) and RM83-10.1.1(5).) 18.p

Wording Changes from Ada 95

{*AI95-00318-02*} Extended_return_statement (see 6.5) is added to the list of constructs that have a declarative region. 18.q/2

Extensions to Ada 2012

{*AI12-0094-1*} **Corrigendum:** access_definition is added to the list of constructs that have a declarative region. This allows parameter names declared in anonymous access type subprogram types to be the same as other names declared outside. For instance: 18.r/4

```
type Foo is record                                              18.s/4
   A : Natural;
   B : access procedure (A : Boolean);
end record;
```

{*AI12-0005-1*} This is now legal, as one would expect; it was illegal in previous versions of Ada as the parameter A and the component A were homographs in the same declarative region (see 8.3). Note that some implementations already allow this common sense interpretation, so this extension might in fact already be used in existing code. 18.t/5

Wording Changes from Ada 2012

{*AI12-0061-1*} {*AI12-0236-1*} {*AI12-0308-1*} Added iterated_component_association, iterated_element_association, and declare_expression to the rapidly expanding list of constructs that have a declarative region. 18.u/5

8.2 Scope of Declarations

[For each declaration, the language rules define a certain portion of the program text called the *scope* of the declaration. The scope of a declaration is also called the scope of any view or entity declared by the 1

declaration. Within the scope of an entity, and only there, there are places where it is legal to refer to the declared entity. These places are defined by the rules of visibility and overloading.]

Static Semantics

2 The *immediate scope* of a declaration is a portion of the declarative region immediately enclosing the declaration. The immediate scope starts at the beginning of the declaration, except in the case of an overloadable declaration, in which case the immediate scope starts just after the place where the profile of the callable entity is determined (which is at the end of the _specification for the callable entity, or at the end of the generic_instantiation if an instance). The immediate scope extends to the end of the declarative region, with the following exceptions:

2.a **Reason:** The reason for making overloadable declarations with profiles special is to simplify compilation: until the compiler has determined the profile, it doesn't know which other declarations are homographs of this one, so it doesn't know which ones this one should hide. Without this rule, two passes over the _specification or generic_instantiation would be required to resolve names that denote things with the same name as this one.

3 - The immediate scope of a library_item includes only its semantic dependents.

3.a/3 **Reason:** {*AI05-0299-1*} Clause 10 defines only a partial ordering of library_items. Therefore, it is a good idea to restrict the immediate scope (and the scope, defined below) to semantic dependents.

3.b Consider also examples like this:

3.c
```
package P is end P;
```

3.d
```
package P.Q is
    I : Integer := 0;
end P.Q;
```

3.e/1
```
with P;
package R is
    package X renames P;
    J : Integer := X.Q.I; -- Illegal!
end R;
```

3.f The scope of P.Q does not contain R. Hence, neither P.Q nor X.Q are visible within R. However, the name R.X.Q would be visible in some other library unit where both R and P.Q are visible (assuming R were made legal by removing the offending declaration).

3.g/2 **Ramification:** {*AI95-00217-06*} This rule applies to limited views as well as “normal” library items. In that case, the semantic dependents are the units that have a limited_with_clause for the limited view.

4 - The immediate scope of a declaration in the private part of a library unit does not include the visible part of any public descendant of that library unit.

4.a **Ramification:** In other words, a declaration in the private part can be visible within the visible part, private part and body of a private child unit. On the other hand, such a declaration can be visible within only the private part and body of a public child unit.

4.b **Reason:** The purpose of this rule is to prevent children from giving private information to clients.

4.c/2 **Ramification:** {*AI95-00231-01*} For a public child subprogram, this means that the parent's private part is not visible in the profile of the declaration and of the body. This is true even for subprogram_bodies that are not completions. For a public child generic unit, it means that the parent's private part is not visible in the generic_formal_part, as well as in the first list of basic_declarative_items (for a generic package), or the (syntactic) profile (for a generic subprogram).

5 [The *visible part* of (a view of) an entity is a portion of the text of its declaration containing declarations that are visible from outside.] The *private part* of (a view of) an entity that has a visible part contains all declarations within the declaration of (the view of) the entity, except those in the visible part; [these are not visible from outside. Visible and private parts are defined only for these kinds of entities: callable entities, other program units, and composite types.]

6 - The visible part of a view of a callable entity is its profile.

7 - The visible part of a composite type other than a task or protected type consists of the declarations of all components declared [(explicitly or implicitly)] within the type_declaration.

- The visible part of a generic unit includes the generic_formal_part. For a generic package, it also includes the first list of basic_declarative_items of the package_specification. For a generic subprogram, it also includes the profile. 8

  Reason: Although there is no way to reference anything but the formals from outside a generic unit, they are still in the visible part in the sense that the corresponding declarations in an instance can be referenced (at least in some cases). In other words, these declarations have an effect on the outside world. The visible part of a generic unit needs to be defined this way in order to properly support the rule that makes a parent's private part invisible within a public child's visible part. 8.a

  Ramification: The visible part of an instance of a generic unit is as defined for packages and subprograms; it is not defined in terms of the visible part of a generic unit. 8.b

- [The visible part of a package, task unit, or protected unit consists of declarations in the program unit's declaration other than those following the reserved word **private**, if any; see 7.1 and 12.7 for packages, 9.1 for task units, and 9.4 for protected units.] 9

The scope of a declaration always contains the immediate scope of the declaration. In addition, for a given declaration that occurs immediately within the visible part of an outer declaration, or is a public child of an outer declaration, the scope of the given declaration extends to the end of the scope of the outer declaration, except that the scope of a library_item includes only its semantic dependents. 10

Ramification: Note the recursion. If a declaration appears in the visible part of a library unit, its scope extends to the end of the scope of the library unit, but since that only includes dependents of the declaration of the library unit, the scope of the inner declaration also only includes those dependents. If X renames library package P, which has a child Q, a with_clause mentioning P.Q is necessary to be able to refer to X.Q, even if P.Q is visible at the place where X is declared. 10.a

{*AI95-00408-01*} {*AI05-0183-1*} The scope of an attribute_definition_clause is identical to the scope of a declaration that would occur at the point of the attribute_definition_clause. The scope of an aspect_specification is identical to the scope of the associated declaration. 10.1/3

The immediate scope of a declaration is also the immediate scope of the entity or view declared by the declaration. Similarly, the scope of a declaration is also the scope of the entity or view declared by the declaration. 11

Ramification: The rule for immediate scope implies the following: 11.a

- If the declaration is that of a library unit, then the immediate scope includes the declarative region of the declaration itself, but not other places, unless they are within the scope of a with_clause that mentions the library unit. 11.b

  It is necessary to attach the semantics of with_clauses to [immediate] scopes (as opposed to visibility), in order for various rules to work properly. A library unit should hide a homographic implicit declaration that appears in its parent, but only within the scope of a with_clause that mentions the library unit. Otherwise, we would violate the "legality determinable via semantic dependences" rule of 10, “Program Structure and Compilation Issues”. The declaration of a library unit should be allowed to be a homograph of an explicit declaration in its parent's body, so long as that body does not mention the library unit in a with_clause. 11.c

  This means that one cannot denote the declaration of the library unit, but one might still be able to denote the library unit via another view. 11.d

  A with_clause does not make the declaration of a library unit visible; the lack of a with_clause prevents it from being visible. Even if a library unit is mentioned in a with_clause, its declaration can still be hidden. 11.e

- The completion of the declaration of a library unit (assuming that's also a declaration) is not visible, neither directly nor by selection, outside that completion. 11.f

- The immediate scope of a declaration immediately within the body of a library unit does not include any child of that library unit. 11.g

  This is needed to prevent children from looking inside their parent's body. The children are in the declarative region of the parent, and they might be after the parent's body. Therefore, the scope of a declaration that occurs immediately within the body might include some children. 11.h

{*AI12-0003-1*} The immediate scope of a pragma that is not used as a configuration pragma is defined to be the region extending from immediately after the pragma to the end of the declarative region immediately enclosing the pragma. 12/4

13/3 NOTE {*AI05-0299-1*} There are notations for denoting visible declarations that are not directly visible. For example, parameter_specifications are in the visible part of a subprogram_declaration so that they can be used in named-notation calls appearing outside the called subprogram. For another example, declarations of the visible part of a package can be denoted by expanded names appearing outside the package, and can be made directly visible by a use_clause.

13.a/2 **Ramification:** {*AI95-00114-01*} There are some obscure cases involving generics in which there is no such notation. See Clause 12.

Extensions to Ada 83

13.b The fact that the immediate scope of an overloadable declaration does not include its profile is new to Ada 95. It replaces RM83-8.3(16), which said that within a subprogram specification and within the formal part of an entry declaration or accept statement, all declarations with the same designator as the subprogram or entry were hidden from all visibility. The RM83-8.3(16) rule seemed to be overkill, and created both implementation difficulties and unnecessary semantic complexity.

Wording Changes from Ada 83

13.c We no longer need to talk about the scope of notations, identifiers, character_literals, and operator_symbols.

13.d/3 {*AI05-0299-1*} The notion of "visible part" has been extended in Ada 95. The syntax of task and protected units now allows private parts, thus requiring us to be able to talk about the visible part as well. It was necessary to extend the concept to subprograms and to generic units, in order for the visibility rules related to child library units to work properly. It was necessary to define the concept separately for generic formal packages, since their visible part is slightly different from that of a normal package. Extending the concept to composite types made the definition of scope slightly simpler. We define visible part for some things elsewhere, since it makes a big difference to the user for those things. For composite types and subprograms, however, the concept is used only in arcane visibility rules, so we localize it to this subclause.

13.e In Ada 83, the semantics of with_clauses was described in terms of visibility. It is now described in terms of [immediate] scope.

13.f We have clarified that the following is illegal (where Q and R are library units):

13.g
```
package Q is
    I : Integer := 0;
end Q;
```

13.h
```
package R is
    package X renames Standard;
    X.Q.I := 17; -- Illegal!
end R;
```

13.i even though Q is declared in the declarative region of Standard, because R does not mention Q in a with_clause.

Wording Changes from Ada 95

13.j/2 {*AI95-00408-01*} The scope of an attribute_definition_clause is defined so that it can be used to define the visibility of such a clause, so *that* can be used by the stream attribute availability rules (see 13.13.2).

Wording Changes from Ada 2005

13.k/3 {*AI05-0183-1*} The scope of an aspect_specification is defined for similar reasons that it was defined for attribute_definition_clauses.

Wording Changes from Ada 2012

13.l/4 {*AI12-0003-1*} The immediate scope of a pragma is defined as it is used in other rules in the Reference Manual.

8.3 Visibility

1 [The *visibility rules*, given below, determine which declarations are visible and directly visible at each place within a program. The visibility rules apply to both explicit and implicit declarations.]

Static Semantics

2 A declaration is defined to be *directly visible* at places where a name consisting of only an identifier or operator_symbol is sufficient to denote the declaration; that is, no selected_component notation or special context (such as preceding => in a named association) is necessary to denote the declaration. A

declaration is defined to be *visible* wherever it is directly visible, as well as at other places where some name (such as a selected_component) can denote the declaration.

The syntactic category direct_name is used to indicate contexts where direct visibility is required. The syntactic category selector_name is used to indicate contexts where visibility, but not direct visibility, is required. 3

There are two kinds of direct visibility: *immediate visibility* and *use-visibility*. A declaration is immediately visible at a place if it is directly visible because the place is within its immediate scope. A declaration is use-visible if it is directly visible because of a use_clause (see 8.4). Both conditions can apply. 4

A declaration can be *hidden*, either from direct visibility, or from all visibility, within certain parts of its scope. Where *hidden from all visibility*, it is not visible at all (neither using a direct_name nor a selector_name). Where *hidden from direct visibility*, only direct visibility is lost; visibility using a selector_name is still possible. 5

[Two or more declarations are *overloaded* if they all have the same defining name and there is a place where they are all directly visible.] 6

Ramification: Note that a name can have more than one possible interpretation even if it denotes a nonoverloadable entity. For example, if there are two functions F that return records, both containing a component called C, then the name F.C has two possible interpretations, even though component declarations are not overloadable. 6.a

The declarations of callable entities [(including enumeration literals)] are *overloadable*[, meaning that overloading is allowed for them]. 7

Ramification: A generic_declaration is not overloadable within its own generic_formal_part. This follows from the rules about when a name denotes a current instance. See AI83-00286. This implies that within a generic_formal_part, outer declarations with the same defining name are hidden from direct visibility. It also implies that if a generic formal parameter has the same defining name as the generic itself, the formal parameter hides the generic from direct visibility. 7.a

Two declarations are *homographs* if they have the same defining name, and, if both are overloadable, their profiles are type conformant. [An inner declaration hides any outer homograph from direct visibility.] 8

Term entry: overriding operation — operation that replaces an inherited primitive operation 8.a/5
Note: Operations can be marked explicitly as overriding or not overriding.

{*8652/0025*} {*AI95-00044-01*} [Two homographs are not generally allowed immediately within the same declarative region unless one *overrides* the other (see Legality Rules below).] The only declarations that are *overridable* are the implicit declarations for predefined operators and inherited primitive subprograms. A declaration overrides another homograph that occurs immediately within the same declarative region in the following cases: 9/1

- {*8652/0025*} {*AI95-00044-01*} A declaration that is not overridable overrides one that is overridable, [regardless of which declaration occurs first]; 10/1

  Ramification: {*8652/0025*} {*AI95-00044-01*} And regardless of whether the nonoverridable declaration is overloadable or not. For example, statement_identifiers are covered by this rule. 10.a/1

  The "regardless of which declaration occurs first" is there because the explicit declaration could be a primitive subprogram of a partial view, and then the full view might inherit a homograph. We are saying that the explicit one wins (within its scope), even though the implicit one comes later. 10.b

  If the overriding declaration is also a subprogram, then it is a primitive subprogram. 10.c

  As explained in 7.3.1, "Private Operations", some inherited primitive subprograms are never declared. Such subprograms cannot be overridden, although they can be reached by dispatching calls in the case of a tagged type. 10.d

- The implicit declaration of an inherited operator overrides that of a predefined operator; 11

11.a **Ramification:** In a previous version of Ada 9X, we tried to avoid the notion of predefined operators, and say that they were inherited from some magical root type. However, this seemed like too much mechanism. Therefore, a type can have a predefined "+" as well as an inherited "+". The above rule says the inherited one wins.

11.b/2 {*AI95-00114-01*} The "regardless of which declaration occurs first" applies here as well, in the case where derived_type_definition in the visible part of a public library unit derives from a private type declared in the parent unit, and the full view of the parent type has additional predefined operators, as explained in 7.3.1, "Private Operations". Those predefined operators can be overridden by inherited subprograms implicitly declared earlier.

12 - An implicit declaration of an inherited subprogram overrides a previous implicit declaration of an inherited subprogram.

12.1/2 - {*AI95-00251-01*} If two or more homographs are implicitly declared at the same place:

12.2/2 - {*AI95-00251-01*} If at least one is a subprogram that is neither a null procedure nor an abstract subprogram, and does not require overriding (see 3.9.3), then they override those that are null procedures, abstract subprograms, or require overriding. If more than one such homograph remains that is not thus overridden, then they are all hidden from all visibility.

12.3/5 - {*AI95-00251-01*} {*AI12-0183-1*} Otherwise (all are null procedures, abstract subprograms, or require overriding), then any null procedure overrides all abstract subprograms and all subprograms that require overriding; if more than one such homograph remains that is not thus overridden, then if the profiles of the remaining homographs are all fully conformant with one another, one is chosen arbitrarily; if not, they are all hidden from all visibility.

12.a/2 **Discussion:** In the case where the implementation arbitrarily chooses one overrider from among a group of inherited subprograms, users should not be able to determine which member was chosen, as the set of inherited subprograms which are chosen from must be fully conformant. This rule is needed in order to allow

12.b/2
```
package Outer is
   package P1 is
      type Ifc1 is interface;
      procedure Null_Procedure (X : Ifc1) is null;
      procedure Abstract_Subp  (X : Ifc1) is abstract;
   end P1;
```

12.c/2
```
   package P2 is
      type Ifc2 is interface;
      procedure Null_Procedure (X : Ifc2) is null;
      procedure Abstract_Subp  (X : Ifc2) is abstract;
   end P2;
```

12.d/2
```
   type T is abstract new P1.Ifc1 and P2.Ifc2 with null record;
end Outer;
```

12.e/2 without requiring that T explicitly override any of its inherited operations.

12.f/2 Full conformance is required here, as we cannot allow the parameter names to differ. If they did differ, the routine which was selected for overriding could be determined by using named parameter notation in a call.

12.g/5 {*AI12-0005-1*} When the subprograms do not conform, we chose not to adopt the "use clause" rule which would make them all visible resulting in likely ambiguity. If we had used such a rule, any successful calls would be confusing; and the fact that there are no Beaujolais-like effects to worry about means we can consider other rules. The hidden-from-all-visibility homographs are still inherited by further derivations, which avoids order-of-declaration dependencies and other anomalies.

12.h/2 We have to be careful to not include arbitrary selection if the routines have real bodies. (This can happen in generics, see the example in the incompatibilities section below.) We don't want the ability to successfully call routines where the body executed depends on the compiler or a phase of the moon.

12.i/2 Note that if the type is concrete, abstract subprograms are inherited as subprograms that require overriding. We include functions that require overriding as well; these don't have real bodies, so they can use the more liberal rules.

13 - [For an implicit declaration of a primitive subprogram in a generic unit, there is a copy of this declaration in an instance.] However, a whole new set of primitive subprograms is implicitly declared for each type declared within the visible part of the instance. These new declarations occur immediately after the type declaration, and override the copied ones. [The copied ones can be called only from within the instance; the new ones can be called only from outside the instance, although for tagged types, the body of a new one can be executed by a call to an old one.]

Discussion: In addition, this is also stated redundantly (again), and is repeated, in 12.3, “Generic Instantiation”. The rationale for the rule is explained there. 13.a

To be honest: {*AI05-0042-1*} The implicit subprograms declared when an operation of a progenitor is implemented by an entry or subprogram also override the appropriate implicitly declared inherited operations of the progenitor. 13.b/3

A declaration is visible within its scope, except where hidden from all visibility, as follows: 14

- An overridden declaration is hidden from all visibility within the scope of the overriding declaration. 15

  Ramification: We have to talk about the scope of the overriding declaration, not its visibility, because it hides even when it is itself hidden. 15.a

  Note that the scope of an explicit subprogram_declaration does not start until after its profile. 15.b

- A declaration is hidden from all visibility until the end of the declaration, except: 16

  - For a record type or record extension, the declaration is hidden from all visibility only until the reserved word **record**; 17

  - {*AI95-00345-01*} {*AI05-0177-1*} For a package_declaration, generic_package_declaration, subprogram_body, or expression_function_declaration, the declaration is hidden from all visibility only until the reserved word **is** of the declaration; 18/3

    Ramification: We're talking about the **is** of the construct itself, here, not some random **is** that might appear in a generic_formal_part. 18.a

  - {*AI95-00345-01*} For a task declaration or protected declaration, the declaration is hidden from all visibility only until the reserved word **with** of the declaration if there is one, or the reserved word **is** of the declaration if there is no **with**. 18.1/2

    To be honest: If there is neither a **with** nor **is**, then the exception does not apply and the name is hidden from all visibility until the end of the declaration. This oddity was inherited from Ada 95. 18.b/2

    Reason: We need the “**with** or **is**” rule so that the visibility within an interface_list does not vary by construct. That would make it harder to complete private extensions and would complicate implementations. 18.c/2

- If the completion of a declaration is a declaration, then within the scope of the completion, the first declaration is hidden from all visibility. Similarly, a discriminant_specification or parameter_specification is hidden within the scope of a corresponding discriminant_specification or parameter_specification of a corresponding completion, or of a corresponding accept_statement. 19

  Ramification: This rule means, for example, that within the scope of a full_type_declaration that completes a private_type_declaration, the name of the type will denote the full_type_declaration, and therefore the full view of the type. On the other hand, if the completion is not a declaration, then it doesn't hide anything, and you can't denote it. 19.a

- {*AI95-00217-06*} {*AI95-00412-01*} The declaration of a library unit (including a library_unit_renaming_declaration) is hidden from all visibility at places outside its declarative region that are not within the scope of a nonlimited_with_clause that mentions it. The limited view of a library package is hidden from all visibility at places that are not within the scope of a limited_with_clause that mentions it; in addition, the limited view is hidden from all visibility within the declarative region of the package, as well as within the scope of any nonlimited_with_clause that mentions the package. Where the declaration of the limited view of a package is visible, any name that denotes the package denotes the limited view, including those provided by a package renaming. 20/2

  Discussion: {*AI95-00217-06*} This is the rule that prevents with_clauses from being transitive; the [immediate] scope includes indirect semantic dependents. This rule also prevents the limited view of a package from being visible in the same place as the full view of the package, which prevents various ripple effects. 20.a/2

- {*AI95-00217-06*} {*AI95-00412-01*} [For each declaration or renaming of a generic unit as a child of some parent generic package, there is a corresponding declaration nested immediately within each instance of the parent.] Such a nested declaration is hidden from all visibility except at places that are within the scope of a with_clause that mentions the child. 20.1/2

21 A declaration with a defining_identifier or defining_operator_symbol is immediately visible [(and hence directly visible)] within its immediate scope except where hidden from direct visibility, as follows:

22 • A declaration is hidden from direct visibility within the immediate scope of a homograph of the declaration, if the homograph occurs within an inner declarative region;

23 • A declaration is also hidden from direct visibility where hidden from all visibility.

23.1/3 {*AI95-00195-01*} {*AI95-00408-01*} {*AI05-0183-1*} An attribute_definition_clause or an aspect_specification is *visible* everywhere within its scope.

Name Resolution Rules

24 A direct_name shall resolve to denote a directly visible declaration whose defining name is the same as the direct_name. A selector_name shall resolve to denote a visible declaration whose defining name is the same as the selector_name.

24.a **Discussion:** "The same as" has the obvious meaning here, so for +, the possible interpretations are declarations whose defining name is "+" (an operator_symbol).

25 These rules on visibility and direct visibility do not apply in a context_clause, a parent_unit_name, or a pragma that appears at the place of a compilation_unit. For those contexts, see the rules in 10.1.6, “Environment-Level Visibility Rules”.

25.a **Ramification:** Direct visibility is irrelevant for character_literals. In terms of overload resolution character_literals are similar to other literals, like **null** — see 4.2. For character_literals, there is no need to worry about hiding, since there is no way to declare homographs.

Legality Rules

26/2 {*8652/0025*} {*8652/0026*} {*AI95-00044-01*} {*AI95-00150-01*} {*AI95-00377-01*} A nonoverridable declaration is illegal if there is a homograph occurring immediately within the same declarative region that is visible at the place of the declaration, and is not hidden from all visibility by the nonoverridable declaration. In addition, a type extension is illegal if somewhere within its immediate scope it has two visible components with the same name. Similarly, the context_clause for a compilation unit is illegal if it mentions (in a with_clause) some library unit, and there is a homograph of the library unit that is visible at the place of the compilation unit, and the homograph and the mentioned library unit are both declared immediately within the same declarative region. These rules also apply to dispatching operations declared in the visible part of an instance of a generic unit. However, they do not apply to other overloadable declarations in an instance[; such declarations may have type conformant profiles in the instance, so long as the corresponding declarations in the generic were not type conformant].

26.a **Discussion:** Normally, these rules just mean you can't explicitly declare two homographs immediately within the same declarative region. The wording is designed to handle the following special cases:

26.b • If the second declaration completes the first one, the second declaration is legal.

26.c • If the body of a library unit contains an explicit homograph of a child of that same library unit, this is illegal only if the body mentions the child in its context_clause, or if some subunit mentions the child. Here's an example:

26.d
```
package P is
end P;
```

26.e
```
package P.Q is
end P.Q;
```

26.f
```
package body P is
    Q : Integer; -- OK; we cannot see package P.Q here.
    procedure Sub is separate;
end P;
```

```
with P.Q;
separate(P)
procedure Sub is -- Illegal.
begin
    null;
end Sub;
```

26.g

If package body P said "**with** P.Q;", then it would be illegal to declare the homograph Q: Integer. But it does not, so the body of P is OK. However, the subunit would be able to see both P.Q's, and is therefore illegal.

26.h

A previous version of Ada 9X allowed the subunit, and said that references to P.Q would tend to be ambiguous. However, that was a bad idea, because it requires overload resolution to resolve references to directly visible nonoverloadable homographs, which is something compilers have never before been required to do.

26.i

- {*8652/0026*} {*8652/0102*} {*AI95-00150-01*} {*AI95-00157-01*} If a type extension contains a component with the same name as a component in an ancestor type, there must be no place where both components are visible. For instance:

26.i.1/1

```
package A is
   type T is tagged private;
   package B is
      type NT is new T with record
         I: Integer; -- Illegal because T.I is visible in the body.
      end record; -- T.I is not visible here.
   end B;
private
   type T is tagged record
      I: Integer; -- Illegal because T.I is visible in the body.
   end record;
end A;
```

26.i.2/1

```
{AI95-00114-01} package body A is
   package body B is
      -- T.I becomes visible here.
   end B;
end A;
```

26.i.3/2

```
package A.C is
   type NT2 is new A.T with record
      I: Integer; -- Illegal because T.I is visible in the private part.
   end record; -- T.I is not visible here.
private
    -- T.I is visible here.
end A.C;
```

26.i.4/1

```
with A;
package D is
   type NT3 is new A.T with record
      I: Integer; -- Legal because T.I is never visible in this package.
   end record;
end D;
```

26.i.5/1

```
with D;
package A.E is
   type NT4 is new D.NT3 with null record;
   X : NT4;
   I1 : Integer := X.I;         -- D.NT3.I
   I2 : Integer := D.NT3(X).I; -- D.NT3.I
   I3 : Integer := A.T(X).I;    -- A.T.I
end A.E;
```

26.i.6/1

{*8652/0102*} {*AI95-00157-01*} D.NT3 can have a component I because the component I of the parent type is never visible. The parent component exists, of course, but is never declared for the type D.NT3. In the child package A.E, the component I of A.T is visible, but that does not change the fact that the A.T.I component was never declared for type D.NT3. Thus, A.E.NT4 does not (visibly) inherit the component I from A.T, while it does inherit the component I from D.NT3. Of course, both components exist, and can be accessed by a type conversion as shown above. This behavior stems from the fact that every characteristic of a type (including components) must be declared somewhere in the innermost declarative region containing the type — if the characteristic is never visible in that declarative region, it is never declared. Therefore, such characteristics do not suddenly become available even if they are in fact visible in some other scope. See 7.3.1 for more on the rules.

26.i.7/1

- {*AI95-00377-01*} It is illegal to mention both an explicit child of an instance, and a child of the generic from which the instance was instantiated. This is easier to understand with an example:

26.i.8/2

```
generic
package G1 is
end G1;
```

26.i.9/2

26.i.10/2
```
generic
package G1.G2 is
end G1.G2;
```

26.i.11/2
```
with G1;
package I1 is new G1;
```

26.i.12/2
```
package I1.G2 renames ...
```

26.i.13/2
```
with G1.G2;
with I1.G2;              -- Illegal
package Bad is ...
```

26.i.14/5 {*AI12-0300-1*} The context clause for Bad is illegal as I1 has an implicit declaration of I1.G2 based on the generic child G1.G2, as well as the mention of the explicit child I1.G2. As in the previous cases, this is illegal only if the context clause makes both children visible; the explicit child can be mentioned as long as the generic child is not (and vice versa).

26.j Note that we need to be careful which things we make "hidden from all visibility" versus which things we make simply illegal for names to denote. The distinction is subtle. The rules that disallow names denoting components within a type declaration (see 3.7) do not make the components invisible at those places, so that the above rule makes components with the same name illegal. The same is true for the rule that disallows names denoting formal parameters within a formal_part (see 6.1).

26.k **Discussion:** The part about instances is from AI83-00012. The reason it says “overloadable declarations” is because we don't want it to apply to type extensions that appear in an instance; components are not overloadable.

27 NOTE 1 Visibility for compilation units follows from the definition of the environment in 10.1.4, except that it is necessary to apply a with_clause to obtain visibility to a library_unit_declaration or library_unit_renaming_declaration.

28 NOTE 2 In addition to the visibility rules given above, the meaning of the occurrence of a direct_name or selector_name at a given place in the text can depend on the overloading rules (see 8.6).

29 NOTE 3 Not all contexts where an identifier, character_literal, or operator_symbol are allowed require visibility of a corresponding declaration. Contexts where visibility is not required are identified by using one of these three syntactic categories directly in a syntax rule, rather than using direct_name or selector_name.

29.a **Ramification:** An identifier, character_literal or operator_symbol that occurs in one of the following contexts is not required to denote a visible or directly visible declaration:

29.b 1. A defining name.

29.c 2. The identifiers or operator_symbol that appear after the reserved word **end** in a proper_body. Similarly for “**end loop**”, etc.

29.d 3. An attribute_designator.

29.e 4. A pragma identifier.

29.f 5. A *pragma_argument_*identifier.

29.g 6. An identifier specific to a pragma used in a pragma argument.

29.g.1/3 7. {*AI05-0183-1*} An aspect_mark;

29.g.2/3 8. {*AI05-0183-1*} An identifier specific to an aspect used in an aspect_definition.

29.h The visibility rules have nothing to do with the above cases; the meanings of such things are defined elsewhere. Reserved words are not identifiers; the visibility rules don't apply to them either.

29.i Because of the way we have defined "declaration", it is possible for a usage name to denote a subprogram_body, either within that body, or (for a nonlibrary unit) after it (since the body hides the corresponding declaration, if any). Other bodies do not work that way. Completions of type_declarations and deferred constant declarations do work that way. Accept_statements are never denoted, although the parameter_specifications in their profiles can be.

29.j The scope of a subprogram does not start until after its profile. Thus, the following is legal:

29.k
```
X : constant Integer := 17;
...
package P is
    procedure X(Y : in Integer := X);
end P;
```

29.l The body of the subprogram will probably be illegal, however, since the constant X will be hidden by then.

29.m The rule is different for generic subprograms, since they are not overloadable; the following is illegal:

```
X : constant Integer := 17;                                          29.n
package P is
    generic
      Z : Integer := X; -- Illegal!
    procedure X(Y : in Integer := X); -- Illegal!
end P;
```

The constant X is hidden from direct visibility by the generic declaration. 29.o

Extensions to Ada 83

Declarations with the same defining name as that of a subprogram or entry being defined are nevertheless visible within the subprogram specification or entry declaration. 29.p

Wording Changes from Ada 83

The term “visible by selection” is no longer defined. We use the terms “directly visible” and “visible” (among other things). There are only two regions of text that are of interest, here: the region in which a declaration is visible, and the region in which it is directly visible. 29.q

Visibility is defined only for declarations. 29.r

Incompatibilities With Ada 95

{*AI95-00251-01*} Added rules to handle the inheritance and overriding of multiple homographs for a single type declaration, in order to support multiple inheritance from interfaces. The new rules are intended to be compatible with the existing rules so that programs that do not use interfaces do not change their legality. However, there is a very rare case where this is not true: 29.s/2

```
generic                                                              29.t/2
   type T1 is private;
   type T2 is private;
package G is
   type T is null record;
   procedure P (X : T; Y : T1);
   procedure P (X : T; Z : T2);
end G;]
```

```
package I is new G (Integer, Integer); -- Exports homographs of P.   29.u/2
type D is new I.T; -- Both Ps are inherited.                         29.v/2
Obj : D;                                                             29.w/2
P (Obj, Z => 10); -- Legal in Ada 95, illegal in Ada 2005.           29.x/2
```

The call to P would resolve in Ada 95 by using the parameter name, while the procedures P would be hidden from all visibility in Ada 2005 and thus would not resolve. This case doesn't seem worth making the rules any more complex than they already are. 29.y/2

{*AI95-00377-01*} **Amendment Correction:** A with_clause is illegal if it would create a homograph of an implicitly declared generic child (see 10.1.1). An Ada 95 compiler could have allowed this, but which unit of the two units involved would be denoted wasn't specified, so any successful use isn't portable. Removing one of the two with_clauses involved will fix the problem. 29.z/2

Wording Changes from Ada 95

{*8652/0025*} {*AI95-00044-01*} **Corrigendum:** Clarified the overriding rules so that "/=" and statement_identifiers are covered. 29.aa/2

{*8652/0026*} {*AI95-00150-01*} **Corrigendum:** Clarified that is it never possible for two components with the same name to be visible; any such program is illegal. 29.bb/2

{*AI95-00195-01*} {*AI95-00408-01*} The visibility of an attribute_definition_clause is defined so that it can be used by the stream attribute availability rules (see 13.13.2). 29.cc/2

{*AI95-00217-06*} The visibility of a limited view of a library package is defined (see 10.1.1). 29.dd/2

Wording Changes from Ada 2005

{*AI05-0177-1*} {*AI12-0157-1*} Added wording so that the parameters of an expression_function_declaration are visible in the return expression of the function. (It would be pretty useless without such a rule.) 29.ee/4

{*AI05-0183-1*} The visibility of an aspect_specification is defined so that it can be used in various other rules. 29.ff/3

8.3.1 Overriding Indicators

1/2 {*AI95-00218-03*} An overriding_indicator is used to declare that an operation is intended to override (or not override) an inherited operation.

Syntax

2/2 {*AI95-00218-03*} overriding_indicator ::= [**not**] **overriding**

Legality Rules

3/3 {*AI95-00218-03*} {*AI95-00348-01*} {*AI95-00397-01*} {*AI05-0177-1*} If an abstract_subprogram_declaration, null_procedure_declaration, expression_function_declaration, subprogram_body, subprogram_body_stub, subprogram_renaming_declaration, generic_instantiation of a subprogram, or subprogram_declaration other than a protected subprogram has an overriding_indicator, then:

4/2 - the operation shall be a primitive operation for some type;

5/2 - if the overriding_indicator is **overriding**, then the operation shall override a homograph at the place of the declaration or body;

5.a/3 **To be honest:** {*AI05-0005-1*} This doesn't require that the overriding happen at precisely the place of the declaration or body; it only requires that the region in which the overriding is known to have happened includes this place. That is, the overriding can happen at or before the place of the declaration or body.

6/2 - if the overriding_indicator is **not overriding**, then the operation shall not override any homograph (at any place).

7/2 In addition to the places where Legality Rules normally apply, these rules also apply in the private part of an instance of a generic unit.

7.a/2 **Discussion:** The **overriding** and **not overriding** rules differ slightly. For **overriding**, we want the indicator to reflect the overriding state at the place of the declaration; otherwise the indicator would be "lying". Whether a homograph is implicitly declared after the declaration (see 7.3.1 to see how this can happen) has no impact on this check. However, **not overriding** is different; "lying" would happen if a homograph declared later actually is overriding. So, we require this check to take into account later overridings. That can be implemented either by looking ahead, or by rechecking when additional operations are declared.

7.b/2 The "no lying" rules are needed to prevent a subprogram_declaration and subprogram_body from having contradictory overriding_indicators.

8/2 NOTE {*AI95-00397-01*} Rules for overriding_indicators of task and protected entries and of protected subprograms are found in 9.5.2 and 9.4, respectively.

Examples

9/5 {*AI95-00433-01*} {*AI12-0429-1*} *Example of use of an overriding indicator when declaring a security queue derived from the Queue interface of 3.9.4:*

10/2
```
type Security_Queue is new Queue with record ...;
```

11/2
```
overriding
procedure Append(Q : in out Security_Queue; Person : in Person_Name);
```

12/5
```
{AI12-0178-1} overriding
procedure Remove_First(Q : in out Security_Queue; Person : out Person_Name);
```

13/2
```
overriding
function Cur_Count(Q : in Security_Queue) return Natural;
```

14/2
```
overriding
function Max_Count(Q : in Security_Queue) return Natural;
```

15/2
```
not overriding
procedure Arrest(Q : in out Security_Queue; Person : in Person_Name);
```

{AI12-0429-1} The first four subprogram declarations guarantee that these subprograms will override the four subprograms inherited from the Queue interface. A misspelling in one of these subprograms will be detected at compile time by the implementation. Conversely, the declaration of Arrest guarantees that this is a new operation. 16/5

Discussion: In this case, the subprograms are abstract, so misspellings will get detected anyway. But for other subprograms (especially when deriving from concrete types), the error might never be detected, and a body other than the one the programmer intended might be executed without warning. Thus our new motto: “Overriding indicators — don't derive a type without them!” 16.a/2

Extensions to Ada 95

{*AI95-00218-03*} Overriding_indicators are new. These let the programmer state her overriding intentions to the compiler; if the compiler disagrees, an error will be produced rather than a hard to find bug. 16.b/2

Wording Changes from Ada 2005

{*AI95-0177-1*} Expression functions can have overriding indicators. 16.c/3

8.4 Use Clauses

[A use_package_clause achieves direct visibility of declarations that appear in the visible part of a package; a use_type_clause achieves direct visibility of the primitive operators of a type.] 1

Language Design Principles

If and only if the visibility rules allow P.A, "**use** P;" should make A directly visible (barring name conflicts). This means, for example, that child library units, and generic formals of a formal package whose formal_package_actual_part is (<>), should be made visible by a use_clause for the appropriate package. 1.a

The rules for use_clauses were carefully constructed to avoid so-called *Beaujolais* effects, where the addition or removal of a single use_clause, or a single declaration in a "use"d package, would change the meaning of a program from one legal interpretation to another. 1.b

Syntax

use_clause ::= use_package_clause | use_type_clause 2

use_package_clause ::= **use** *package_*name {, *package_*name}; 3

{*AI05-0150-1*} use_type_clause ::= **use** [**all**] **type** subtype_mark {, subtype_mark}; 4/3

Legality Rules

{*AI95-00217-06*} A *package_*name of a use_package_clause shall denote a nonlimited view of a package. 5/2

Ramification: This includes formal packages. 5.a

Static Semantics

For each use_clause, there is a certain region of text called the *scope* of the use_clause. For a use_clause within a context_clause of a library_unit_declaration or library_unit_renaming_-declaration, the scope is the entire declarative region of the declaration. For a use_clause within a context_clause of a body, the scope is the entire body [and any subunits (including multiply nested subunits). The scope does not include context_clauses themselves.] 6

For a use_clause immediately within a declarative region, the scope is the portion of the declarative region starting just after the use_clause and extending to the end of the declarative region. However, the scope of a use_clause in the private part of a library unit does not include the visible part of any public descendant of that library unit. 7

7.a **Reason:** The exception echoes the similar exception for "immediate scope (of a declaration)" (see 8.2). It makes use_clauses work like this:

7.b
```
package P is
    type T is range 1..10;
end P;
```

7.c
```
with P;
package Parent is
private
    use P;
    X : T;
end Parent;
```

7.d
```
package Parent.Child is
    Y : T; -- Illegal!
    Z : P.T;
private
    W : T;
end Parent.Child;
```

7.e The declaration of Y is illegal because the scope of the "**use** P" does not include that place, so T is not directly visible there. The declarations of X, Z, and W are legal.

7.1/2 {*AI95-00217-06*} A package is *named* in a use_package_clause if it is denoted by a *package_*name of that clause. A type is *named* in a use_type_clause if it is determined by a subtype_mark of that clause.

8/3 {*AI95-00217-06*} {*AI05-0150-1*} For each package named in a use_package_clause whose scope encloses a place, each declaration that occurs immediately within the declarative region of the package is *potentially use-visible* at this place if the declaration is visible at this place. For each type *T* or *T*'Class named in a use_type_clause whose scope encloses a place, the declaration of each primitive operator of type *T* is potentially use-visible at this place if its declaration is visible at this place. If a use_type_clause whose scope encloses a place includes the reserved word **all**, then the following entities are also potentially use-visible at this place if the declaration of the entity is visible at this place:

8.1/3 - {*AI05-0150-1*} Each primitive subprogram of *T* including each enumeration literal (if any);

8.2/3 - {*AI05-0150-1*} Each subprogram that is declared immediately within the declarative region in which an ancestor type of *T* is declared and that operates on a class-wide type that covers *T*.

8.a/3 **Ramification:** {*AI05-0150-1*} Primitive subprograms whose defining name is an identifier are *not* made potentially visible by a use_type_clause unless reserved word **all** is included. A use_type_clause without **all** is only for operators.

8.b The semantics described here should be similar to the semantics for expanded names given in 4.1.3, "Selected Components" so as to achieve the effect requested by the "principle of equivalence of use_clauses and selected_components". Thus, child library units and generic formal parameters of a formal package are potentially use-visible when their enclosing package is use'd.

8.c The "visible at that place" part implies that applying a use_clause to a parent unit does not make all of its children use-visible — only those that have been made visible by a with_clause. It also implies that we don't have to worry about hiding in the definition of "directly visible" — a declaration cannot be use-visible unless it is visible.

8.d Note that "**use type** T'Class;" is equivalent to "**use type** T;", which helps avoid breaking the generic contract model.

8.3/3 {*AI05-0131-1*} Certain implicit declarations may become potentially use-visible in certain contexts as described in 12.6.

9 A declaration is *use-visible* if it is potentially use-visible, except in these naming-conflict cases:

10 - A potentially use-visible declaration is not use-visible if the place considered is within the immediate scope of a homograph of the declaration.

11 - Potentially use-visible declarations that have the same identifier are not use-visible unless each of them is an overloadable declaration.

11.a **Ramification:** Overloadable declarations don't cancel each other out, even if they are homographs, though if they are not distinguishable by formal parameter names or the presence or absence of default_expressions, any use will be ambiguous. We only mention identifiers here, because declarations named by operator_symbols are always overloadable, and hence never cancel each other. Direct visibility is irrelevant for character_literals.

Dynamic Semantics

The elaboration of a use_clause has no effect. 12

Examples

Example of a use clause in a context clause: 13

```
with Ada.Calendar; use Ada;
```
14

Example of a use type clause: 15

```
use type Rational_Numbers.Rational; -- see 7.1
Two_Thirds: Rational_Numbers.Rational := 2/3;
```
16

Ramification: In "**use** X, Y;", Y cannot refer to something made visible by the "**use**" of X. Thus, it's not (quite) equivalent to "**use** X; **use** Y;". 16.a

If a given declaration is already immediately visible, then a use_clause that makes it potentially use-visible has no effect. Therefore, a use_type_clause for a type whose declaration appears in a place other than the visible part of a package has no effect; it cannot make a declaration use-visible unless that declaration is already immediately visible. 16.b

"**Use type** S1;" and "**use type** S2;" are equivalent if S1 and S2 are both subtypes of the same type. In particular, "**use type** S;" and "**use type** S'Base;" are equivalent. 16.c

Reason: We considered adding a rule that prevented several declarations of views of the same entity that all have the same semantics from cancelling each other out. For example, if a (possibly implicit) subprogram_declaration for "+" is potentially use-visible, and a fully conformant renaming of it is also potentially use-visible, then they (annoyingly) cancel each other out; neither one is use-visible. The considered rule would have made just one of them use-visible. We gave up on this idea due to the complexity of the rule. It would have had to account for both overloadable and nonoverloadable renaming_declarations, the case where the rule should apply only to some subset of the declarations with the same defining name, and the case of subtype_declarations (since they are claimed to be sufficient for renaming of subtypes). 16.d

Extensions to Ada 83

The use_type_clause is new to Ada 95. 16.e

Wording Changes from Ada 83

The phrase "omitting from this set any packages that enclose this place" is no longer necessary to avoid making something visible outside its scope, because we explicitly state that the declaration has to be visible in order to be potentially use-visible. 16.f

Wording Changes from Ada 95

{*AI95-00217-06*} Limited views of packages are not allowed in use clauses. Defined *named in a use clause* for use in other limited view rules (see 10.1.2). 16.g/2

Extensions to Ada 2005

{*AI05-0150-1*} The **use all type** version of the use_type_clause is new to Ada 2012. It works similarly to prefixed views. 16.h/3

Wording Changes from Ada 2005

{*AI05-0131-1*} **Correction:** Added wording to allow other declarations to be potentially use-visible, to support corrections to formal subprograms. 16.i/3

8.5 Renaming Declarations

[A renaming_declaration declares another name for an entity, such as an object, exception, package, subprogram, entry, or generic unit. Alternatively, a subprogram_renaming_declaration can be the completion of a previous subprogram_declaration.] 1

Term entry: renaming — declaration that does not define a new entity, but instead defines a new view of an existing entity 1.a/5

Syntax

2 renaming_declaration ::=
    object_renaming_declaration
  | exception_renaming_declaration
  | package_renaming_declaration
  | subprogram_renaming_declaration
  | generic_renaming_declaration

Dynamic Semantics

3 The elaboration of a renaming_declaration evaluates the name that follows the reserved word **renames** and thereby determines the view and entity denoted by this name (the *renamed view* and *renamed entity*). [A name that denotes the renaming_declaration denotes (a new view of) the renamed entity.]

4/5 NOTE 1 {*AI12-0442-1*} Renaming can be used to resolve name conflicts and to act as a shorthand. Renaming with a different identifier or operator_symbol does not hide the old name; the new name and the old name can be visible at different places.

5/5 *This paragraph was deleted.*{*AI12-0427-1*}

6 NOTE 2 A subtype defined without any additional constraint can be used to achieve the effect of renaming another subtype (including a task or protected subtype) as in

7
```
subtype Mode is Ada.Text_IO.File_Mode;
```

Wording Changes from Ada 83

7.a The second sentence of RM83-8.5(3), “At any point where a renaming declaration is visible, the identifier, or operator symbol of this declaration denotes the renamed entity.” is incorrect. It doesn't say directly visible. Also, such an identifier might resolve to something else.

7.b The verbiage about renamings being legal “only if exactly one...”, which appears in RM83-8.5(4) (for objects) and RM83-8.5(7) (for subprograms) is removed, because it follows from the normal rules about overload resolution. For language lawyers, these facts are obvious; for programmers, they are irrelevant, since failing these tests is highly unlikely.

8.5.1 Object Renaming Declarations

1/5 {*AI12-0383-1*} [An object_renaming_declaration is used to rename an object or value.]

Syntax

2/5 {*AI95-00230-01*} {*AI95-00423-01*} {*AI05-0183-1*} {*AI12-0275-1*} object_renaming_declaration ::=
    defining_identifier [: [null_exclusion] subtype_mark] **renames** *object_*name
      [aspect_specification];
  | defining_identifier : access_definition **renames** *object_*name
      [aspect_specification];

Name Resolution Rules

3/5 {*AI95-00230-01*} {*AI95-00254-01*} {*AI95-00409-01*} {*AI12-0275-1*} The type of the *object_*name shall resolve to the type determined by the subtype_mark, if present. If no subtype_mark or access_definition is present, the expected type of the *object_*name is any type.

3.1/5 {*AI12-0275-1*} In the case where the type is defined by an access_definition, the type of the *object_*name shall resolve to an anonymous access type. If the anonymous access type is an access-to-object type, the type of the *object_*name shall have the same designated type as that of the access_definition. If the anonymous access type is an access-to-subprogram type, the type of the *object_*name shall have a designated profile that is type conformant with that of the access_definition.

Reason: A previous version of Ada 9X used the usual "expected type" wording: 3.a
"The expected type for the *object*_name is that determined by the subtype_mark."
We changed it so that this would be illegal:

```
X: T;                                                    3.b
Y: T'Class renames X; -- Illegal!
```

When the above was legal, it was unclear whether Y was of type T or T'Class. Note that we still allow this: 3.c

```
Z: T'Class := ...;                                       3.d
W: T renames F(Z);
```

where F is a function with a controlling parameter and result. This is admittedly a bit odd. 3.e

Note that the matching rule for generic formal parameters of mode **in out** was changed to keep it consistent with the rule for renaming. That makes the rule different for **in** vs. **in out**. 3.f

Legality Rules

{*AI12-0383-1*} The renamed entity shall be an object or value. 4/5

{*AI95-00231-01*} {*AI95-00409-01*} {*AI12-0383-1*} In the case where the type is defined by an access_definition, the type of the renamed entity and the type defined by the access_definition: 4.1/5

- {*AI95-00231-01*} {*AI95-00409-01*} shall both be access-to-object types with statically matching designated subtypes and with both or neither being access-to-constant types; or 4.2/2
- {*AI95-00409-01*} shall both be access-to-subprogram types with subtype conformant designated profiles. 4.3/2

{*AI95-00423-01*} {*AI12-0287-1*} For an object_renaming_declaration with a null_exclusion or an access_definition that has a null_exclusion, the subtype of the *object*_name shall exclude null. In addition, if the object_renaming_declaration occurs within the body of a generic unit *G* or within the body of a generic unit declared within the declarative region of generic unit *G*, then: 4.4/5

- {*AI12-0287-1*} if the *object*_name statically denotes a generic formal object of mode **in out** of *G*, then the declaration of that object shall have a null_exclusion; 4.5/5
- {*AI12-0287-1*} if the *object*_name statically denotes a call of a generic formal function of *G*, then the declaration of the result of that function shall have a null_exclusion. 4.6/5

Reason: {*AI12-0287-1*} {*AI12-0005-1*} These rules prevent "lying". **Null** must never be the value of an object with an explicit null_exclusion. The bullets are assume-the-worst rules that prevent trouble in two obscure cases: 4.a/5

```
type Acc_I is access Integer;                            4.b/2
subtype Acc_NN_I is not null Acc_I;
Obj : Acc_I := null;
```

```
generic                                                  4.c/2
   B : in out Acc_NN_I;
package Gen is
   ...
end Gen;
```

```
package body Gen is                                      4.d/2
   D : not null Acc_I renames B;
end Gen;
```

```
package Inst is new Gen (B => Obj);                      4.e/2
```

{*AI12-0287-1*} Without the first bullet rule, D would be legal, and contain the value **null**, because the rule about lying is satisfied for generic matching (Obj matches B; B does not explicitly state **not null**), Legality Rules are not rechecked in the body of any instance, and the template passes the lying rule as well. The second bullet handles a similar case involving formal functions. The rules are so complex because they have to apply to formals used in bodies of child generics as well as in the bodies of generics. 4.f/5

{*AI12-0401-1*} In the case where the *object*_name is a qualified_expression with a nominal subtype *S* and whose expression is a name that denotes an object *Q*: 4.7/5

- if *S* is an elementary subtype, then: 4.8/5
  - *Q* shall be a constant other than a dereference of an access type; or 4.9/5
  - the nominal subtype of *Q* shall be statically compatible with *S*; or 4.10/5

4.11/5 - *S* shall statically match the base subtype of its type if scalar, or the first subtype of its type if an access type.

4.12/5 - if *S* is a composite subtype, then *Q* shall be known to be constrained or *S* shall statically match the first subtype of its type.

4.f.1/5 **Ramification:** There's no restriction if the expression is a value.

4.f.2/5 **Reason:** This check prevents the renamed object from violating its nominal subtype. As the subtype is only checked when the object is renamed, we make it illegal if the actual object is a variable whose value could be changed afterwards to violate the subtype. This is messy as "known to be constrained" is only defined for composite objects, so we have to handle elementary objects and all values separately.

5/5 {*8652/0017*} {*AI95-00184-01*} {*AI95-00363-01*} {*AI05-0008-1*} {*AI12-0401-1*} The renamed entity shall not be a subcomponent that depends on discriminants of an object whose nominal subtype is unconstrained unless the object is known to be constrained. A slice of an array shall not be renamed if this restriction disallows renaming of the array.

5.1/5 {*AI12-0401-1*} In addition to the places where Legality Rules normally apply (see 12.3), these rules also apply in the private part of an instance of a generic unit.

5.a.1/5 **Discussion:** This applies to all of the Legality Rules in this subclause. Rechecks are needed for most of the rules (but not the first two).

5.a **Reason:** This prevents renaming of subcomponents that might disappear, which might leave dangling references. Similar restrictions exist for the Access attribute.

5.a.1/3 {*8652/0017*} {*AI95-00184-01*} {*AI05-0008-1*} The "recheck on instantiation" requirement on generics is necessary to avoid renaming of components which could disappear even when the nominal subtype would prevent the problem:

5.a.2/1
```
type T1 (D1 : Boolean) is
   record
      case D1 is
         when False =>
            C1 : Integer;
         when True =>
            null;
         end case;
      end record;
```

5.a.3/1
```
generic
   type F is new T1;
   X : in out F;
package G is
   C1_Ren : Integer renames X.C1;
end G;
```

5.a.4/1
```
type T2 (D2 : Boolean := False) is new T1 (D1 => D2);

Y : T2;

package I is new G (T2, Y);

Y := (D1 => True); -- Oops!  What happened to I.C1_Ren?
```

5.a.5/3 {*AI05-0008-1*} In addition, the "known to be constrained" rules include assume-the-worst rules for generic bodies partially to prevent such problems.

5.b **Implementation Note:** Note that if an implementation chooses to deallocate-then-reallocate on assignment_statements assigning to unconstrained definite objects, then it cannot represent renamings and access values as simple addresses, because the above rule does not apply to all components of such an object.

5.c **Ramification:** If it is a generic formal object, then the assume-the-best or assume-the-worst rules are applied as appropriate.

Static Semantics

6/5 {*AI95-00230-01*} {*AI95-00409-01*} {*AI12-0383-1*} An object_renaming_declaration declares a new view [of the renamed entity] whose properties are identical to those of the renamed view. [Thus, the properties of the renamed entity are not affected by the renaming_declaration. In particular, its nominal subtype, whether it is a value or an object, its value if it is an object, and whether or not it is a constant,

are unaffected; similarly, the constraints and other properties of its nominal subtype are not affected by renaming (any constraint implied by the subtype_mark or access_definition of the object_renaming_declaration is ignored).]

Discussion: Because the constraints are ignored, it is a good idea to use the nominal subtype of the renamed object when writing an object_renaming_declaration. 6.a

{*AI95-00409-01*} If no null_exclusion is given in the renaming, the object may or may not exclude null. This is similar to the way that constraints need not match, and **constant** is not specified. The renaming defines a view of the renamed entity, inheriting the original properties. 6.b/2

Examples

Example of renaming an object: 7

```
declare                                                                    8
   L : Person renames Leftmost_Person; -- see 3.10.1
begin
   L.Age := L.Age + 1;
end;
```

{*AI12-0383-1*} *Example of renaming a value:* 9/5

```
{AI12-0383-1} Uno renames One;  -- see 3.3.2                               10/5
```

Wording Changes from Ada 83

The phrase “subtype ... as defined in a corresponding object declaration, component declaration, or component subtype indication”, from RM83-8.5(5), is incorrect in Ada 95; therefore we removed it. It is incorrect in the case of an object with an indefinite unconstrained nominal subtype. 10.a

Incompatibilities With Ada 95

{*AI95-00363-01*} Aliased variables are not necessarily constrained in Ada 2005 (see 3.6). Therefore, a subcomponent of an aliased variable may disappear or change shape, and renaming such a subcomponent thus is illegal, while the same operation would have been legal in Ada 95. Note that most allocated objects are still constrained by their initial value (see 4.8), and thus have no change in the legality of renaming for them. For example, using the type T2 of the previous example: 10.b/2

```
AT2 : aliased T2;                                                          10.c/2
C1_Ren : Integer renames AT2.C1; -- Illegal in Ada 2005, legal in Ada 95
AT2 := (D1 => True);             -- Raised Constraint_Error in Ada 95,
                                 -- but does not in Ada 2005, so C1_Ren becomes
                                 -- invalid when this is assigned.
```

Extensions to Ada 95

{*AI95-00230-01*} {*AI95-00231-01*} {*AI95-00254-01*} {*AI95-00409-01*} A renaming can have an anonymous access type. In that case, the accessibility of the renaming is that of the original object (accessibility is not lost as it is for assignment to a component or stand-alone object). 10.d/2

{*AI95-00231-01*} {*AI95-00423-01*} A renaming can have a null_exclusion; if so, the renamed object must also exclude null, so that the null_exclusion does not lie. On the other hand, if the renaming does not have a null_exclusion. it excludes null if the renamed object does. 10.e/2

Wording Changes from Ada 95

{*8652/0017*} {*AI95-00184-01*} **Corrigendum:** Fixed to forbid renamings of depends-on-discriminant components if the type *might* be definite. 10.f/2

Incompatibilities With Ada 2005

{*AI05-0008-1*} **Correction:** Simplified the description of when a discriminant-dependent component is allowed to be renamed — it's now simply when the object is known to be constrained. This fixes a confusion as to whether a subcomponent of an object that is not certain to be constrained can be renamed. The fix introduces an incompatibility, as the rule did not apply in Ada 95 if the prefix was a constant; but it now applies no matter what kind of object is involved. The incompatibility is not too bad, since most kinds of constants are known to be constrained. 10.g/3

Extensions to Ada 2005

10.h/3 {*AI05-0183-1*} An optional aspect_specification can be used in an object_renaming_declaration. This is described in 13.1.1.

Incompatibilities With Ada 2012

10.i/5 {*AI12-0287-1*} **Correction:** The Legality Rules for renames with null exclusions no longer applies to generic formal objects of mode **in**, but does apply to renames of generic formal functions. This means a few unlikely programs are now illegal that were previously allowed by original Ada 2012, while more programs that were previously llegal will be allowed.

10.j/5 {*AI12-0401-1*} **Correction:** Added a rule to ensure that a renaming of a qualified_expression of a variable is allowed only if the variable will always remain within the nominal subtype of the qualified_expression. This was not required in Ada 2012. Renamings that are now illegal are at risk of causing erroneous execution if the variable value is changed to a bad value; this is consistent with other rules preventing renamings from changing to violate their known properties.

Extensions to Ada 2012

10.k/5 {*AI12-0275-1*} The subtype_mark in an object renaming is now optional, as the subtype information it provides is not trustworthy anyway (that comes from the renamed object and there is no requirement that it is the same as that of the object).

10.l/5 {*AI12-0383-1*} An object renaming can now rename values, such as named numbers. The renamed entity still has to be a name, but an arbitrary expression can be renamed by qualifying it.

8.5.2 Exception Renaming Declarations

1 [An exception_renaming_declaration is used to rename an exception.]

Syntax

2/3 {*AI05-0183-1*} exception_renaming_declaration ::=
defining_identifier : **exception renames** *exception*_name
[aspect_specification];

Legality Rules

3 The renamed entity shall be an exception.

Static Semantics

4 An exception_renaming_declaration declares a new view [of the renamed exception].

Examples

5 *Example of renaming an exception:*

6
```
EOF : exception renames Ada.IO_Exceptions.End_Error; -- see A.13
```

Extensions to Ada 2005

6.a/3 {*AI05-0183-1*} An optional aspect_specification can be used in an exception_renaming_declaration. This is described in 13.1.1.

8.5.3 Package Renaming Declarations

1 [A package_renaming_declaration is used to rename a package.]

Syntax

2/3 {*AI05-0183-1*} package_renaming_declaration ::=
package defining_program_unit_name **renames** *package*_name
[aspect_specification];

Legality Rules

The renamed entity shall be a package. 3

{*AI95-00217-06*} {*AI95-00412-01*} If the *package_*name of a package_renaming_declaration denotes a limited view of a package *P*, then a name that denotes the package_renaming_declaration shall occur only within the immediate scope of the renaming or the scope of a with_clause that mentions the package *P* or, if *P* is a nested package, the innermost library package enclosing *P*. 3.1/2

Discussion: The use of a renaming that designates a limited view is restricted to locations where we know whether the view is limited or nonlimited (based on a with_clause). We don't want to make an implicit limited view, as those are not transitive like a regular view. Implementations should be able to see all limited views needed based on the context_clause. 3.a.1/2

Static Semantics

A package_renaming_declaration declares a new view [of the renamed package]. 4

{*AI95-00412-01*} [At places where the declaration of the limited view of the renamed package is visible, a name that denotes the package_renaming_declaration denotes a limited view of the package (see 10.1.1).] 4.1/2

Proof: This rule is found in 8.3, “Visibility”. 4.a.1/2

Examples

Example of renaming a package: 5

```
package TM renames Table_Manager;
```
6

Wording Changes from Ada 95

{*AI95-00217-06*} {*AI95-00412-01*} Uses of renamed limited views of packages can only be used within the scope of a with_clause for the renamed package. 6.a/2

Extensions to Ada 2005

{*AI05-0183-1*} An optional aspect_specification can be used in a package_renaming_declaration. This is described in 13.1.1. 6.b/3

8.5.4 Subprogram Renaming Declarations

{*AI05-0299-1*} A subprogram_renaming_declaration can serve as the completion of a subprogram_declaration; such a renaming_declaration is called a *renaming-as-body*. A subprogram_renaming_declaration that is not a completion is called a *renaming-as-declaration*[, and is used to rename a subprogram (possibly an enumeration literal) or an entry]. 1/3

Ramification: A renaming-as-body is a declaration, as defined in Clause 3. 1.a

Syntax

{*AI95-00218-03*} {*AI05-0183-1*} subprogram_renaming_declaration ::= 2/3

```
  [overriding_indicator]
  subprogram_specification renames callable_entity_name
    [aspect_specification];
```

Name Resolution Rules

The expected profile for the *callable_entity_*name is the profile given in the subprogram_specification. 3

Legality Rules

4/5 {*AI05-0239-1*} {*AI12-0373-1*} The profile of a renaming-as-declaration shall be mode conformant with that of the renamed callable entity.

4.1/2 {*AI95-00423-01*} For a parameter or result subtype of the subprogram_specification that has an explicit null_exclusion:

4.2/5 - {*AI12-0287-1*} if the *callable_entity*_name statically denotes a generic formal subprogram of a generic unit *G*, and the subprogram_renaming_declaration occurs within the body of a generic unit *G* or within the body of a generic unit declared within the declarative region of the generic unit *G*, then the corresponding parameter or result subtype of the formal subprogram of *G* shall have a null_exclusion;

4.3/2 - otherwise, the subtype of the corresponding parameter or result type of the renamed callable entity shall exclude null. In addition to the places where Legality Rules normally apply (see 12.3), this rule applies also in the private part of an instance of a generic unit.

4.a/2 **Reason:** This rule prevents "lying". **Null** must never be the value of a parameter or result with an explicit null_exclusion. The first bullet is an assume-the-worst rule which prevents trouble in generic bodies (including bodies of child units) when the formal subtype excludes null implicitly.

5/3 {*8652/0027*} {*8652/0028*} {*AI95-00135-01*} {*AI95-00145-01*} {*AI05-0239-1*} The profile of a renaming-as-body shall conform fully to that of the declaration it completes. If the renaming-as-body completes that declaration before the subprogram it declares is frozen, the profile shall be mode conformant with that of the renamed callable entity and the subprogram it declares takes its convention from the renamed subprogram; otherwise, the profile shall be subtype conformant with that of the renamed callable entity and the convention of the renamed subprogram shall not be Intrinsic. A renaming-as-body is illegal if the declaration occurs before the subprogram whose declaration it completes is frozen, and the renaming renames the subprogram itself, through one or more subprogram renaming declarations, none of whose subprograms has been frozen.

5.a/1 **Reason:** The otherwise part of the second sentence is to allow an implementation of a renaming-as-body as a single jump instruction to the target subprogram. Among other things, this prevents a subprogram from being completed with a renaming of an entry. (In most cases, the target of the jump can be filled in at link time. In some cases, such as a renaming of a name like "A(I).**all**", an indirect jump is needed. Note that the name is evaluated at renaming time, not at call time.)

5.a.1/1 {*8652/0028*} {*AI95-00145-01*} The first part of the second sentence is intended to allow renaming-as-body of predefined operators before the subprogram_declaration is frozen. For some types (such as integer types), the parameter type for operators is the base type, and it would be very strange for

```
function Equal (A, B : in T) return Boolean;
function Equal (A, B : in T) return Boolean renames "=";
```

to be illegal. (Note that predefined operators cannot be renamed this way after the subprogram_declaration is frozen, as they have convention Intrinsic.)

5.b/1 The first sentence is the normal rule for completions of subprogram_declarations.

5.c **Ramification:** An entry_declaration, unlike a subprogram_declaration, cannot be completed with a renaming_-declaration. Nor can a generic_subprogram_declaration.

5.d The syntax rules prevent a protected subprogram declaration from being completed by a renaming. This is fortunate, because it allows us to avoid worrying about whether the implicit protected object parameter of a protected operation is involved in the conformance rules.

5.d.1/1 **Reason:** {*8652/0027*} {*AI95-00135-01*} Circular renames before freezing is illegal, as the compiler would not be able to determine the convention of the subprogram. Other circular renames are handled below; see Bounded (Run-Time) Errors.

5.1/2 {*AI95-00228-01*} The *callable_entity*_name of a renaming shall not denote a subprogram that requires overriding (see 3.9.3).

5.d.2/2 **Reason:** {*AI95-00228-01*} Such a rename cannot be of the inherited subprogram (which requires overriding because it cannot be called), and thus cannot squirrel away a subprogram (see below). That would be confusing, so we make it

illegal. The renaming is allowed after the overriding, as then the name will denote the overriding subprogram, not the inherited one.

{*AI95-00228-01*} The *callable_entity_*name of a renaming-as-body shall not denote an abstract subprogram. 5.2/2

Reason: {*AI95-00228-01*} Such a subprogram has no body, so it hardly can replace one in the program. 5.d.3/2

{*AI12-0204-1*} If the *callable_entity_*name of a renaming is a prefixed view, the prefix of that view shall denote an object for which renaming is allowed. 5.3/5

Reason: {*AI12-0204-1*} The prefix in such a case is essentially renamed and passed to any calls of the renamed subprogram. If the prefix isn't legal to rename, that doesn't make sense (and allowing it might end up passing a nonexistent object to some calls). 5.e/5

A name that denotes a formal parameter of the subprogram_specification is not allowed within the *callable_entity_*name. 6

Reason: This is to prevent things like this: 6.a

```
function F(X : Integer) return Integer renames Table(X).all;
```
6.b

A similar rule in 6.1 forbids things like this: 6.c

```
function F(X : Integer; Y : Integer := X) return Integer;
```
6.d

Static Semantics

A renaming-as-declaration declares a new view of the renamed entity. The profile of this new view takes its subtypes, parameter modes, and calling convention from the original profile of the callable entity, while taking the formal parameter names and default_expressions from the profile given in the subprogram_renaming_declaration. The new view is a function or procedure, never an entry. 7

To be honest: When renaming an entry as a procedure, the compile-time rules apply as if the new view is a procedure, but the run-time semantics of a call are that of an entry call. 7.a

Ramification: For example, it is illegal for the entry_call_statement of a timed_entry_call to call the new view. But what looks like a procedure call will do things like barrier waiting. 7.b

{*8652/0105*} {*AI95-00211-01*} {*AI95-00228-01*} {*AI05-0095-1*} All properties of the renamed entity are inherited by the new view unless otherwise stated by this document. In particular, if the renamed entity is abstract, the new view also is abstract. Similarly, if the renamed entity is not a program unit, then neither is the renaming. (Implicitly declared subprograms are not program units, see 10.1). 7.b.1/3

Dynamic Semantics

{*8652/0014*} {*AI95-00064-01*} For a call to a subprogram whose body is given as a renaming-as-body, the execution of the renaming-as-body is equivalent to the execution of a subprogram_body that simply calls the renamed subprogram with its formal parameters as the actual parameters and, if it is a function, returns the value of the call. 7.1/1

Ramification: This implies that the subprogram completed by the renaming-as-body has its own elaboration check. 7.b.2/1

{*AI05-0123-1*} For a call on a renaming of a dispatching subprogram that is overridden, if the overriding occurred before the renaming, then the body executed is that of the overriding declaration, even if the overriding declaration is not visible at the place of the renaming; otherwise, the inherited or predefined subprogram is called. A corresponding rule applies to a call on a renaming of a predefined equality operator for an untagged record type. 8/3

Discussion: Note that whether or not the renaming is itself primitive has nothing to do with the renamed subprogram. 8.a

{*AI05-0123-1*} Note that the above rule is only for tagged types and equality of untagged record types. 8.b/3

8.c Consider the following example:

8.d
```
package P is
    type T is tagged null record;
    function Predefined_Equal(X, Y : T) return Boolean renames "=";
private
    function "="(X, Y : T) return Boolean; -- Override predefined "=".
end P;
```

8.e
```
with P; use P;
package Q is
    function User_Defined_Equal(X, Y : T) return Boolean renames P."=";
end Q;
```

8.f A call on Predefined_Equal will execute the predefined equality operator of T, whereas a call on User_Defined_Equal will execute the body of the overriding declaration in the private part of P.

8.g Thus a renaming allows one to squirrel away a copy of an inherited or predefined subprogram before later overriding it.

Bounded (Run-Time) Errors

8.1/1 {*8652/0027*} {*AI95-00135-01*} If a subprogram directly or indirectly renames itself, then it is a bounded error to call that subprogram. Possible consequences are that Program_Error or Storage_Error is raised, or that the call results in infinite recursion.

8.g.1/1 **Reason:** {*8652/0027*} {*AI95-00135-01*} This has to be a bounded error, as it is possible for a renaming-as-body appearing in a package body to cause this problem. Thus it is not possible in general to detect this problem at compile time.

9 NOTE 1 A procedure can only be renamed as a procedure. A function whose defining_designator is either an identifier or an operator_symbol can be renamed with either an identifier or an operator_symbol; for renaming as an operator, the subprogram specification given in the renaming_declaration is subject to the rules given in 6.6 for operator declarations. Enumeration literals can be renamed as functions; similarly, attribute_references that denote functions (such as references to Succ and Pred) can be renamed as functions. An entry can only be renamed as a procedure; the new name is only allowed to appear in contexts that allow a procedure name. An entry of a family can be renamed, but an entry family cannot be renamed as a whole.

10 NOTE 2 The operators of the root numeric types cannot be renamed because the types in the profile are anonymous, so the corresponding specifications cannot be written; the same holds for certain attributes, such as Pos.

11/5 *This paragraph was deleted.*{*AI12-0292-1*}

12 NOTE 3 The primitiveness of a renaming-as-declaration is determined by its profile, and by where it occurs, as for any declaration of (a view of) a subprogram; primitiveness is not determined by the renamed view. In order to perform a dispatching call, the subprogram name has to denote a primitive subprogram, not a nonprimitive renaming of a primitive subprogram.

12.a **Reason:** A subprogram_renaming_declaration could more properly be called renaming_as_subprogram_-declaration, since you're renaming something as a subprogram, but you're not necessarily renaming a subprogram. But that's too much of a mouthful. Or, alternatively, we could call it a callable_entity_renaming_declaration, but that's even worse. Not only is it a mouthful, it emphasizes the entity being renamed, rather than the new view, which we think is a bad idea. We'll live with the oddity.

Examples

13 *Examples of subprogram renaming declarations:*

14
```
procedure My_Write(C : in Character) renames Pool(K).Write; -- see 4.1.3
```

15
```
function Real_Plus(Left, Right : Real   ) return Real    renames "+";
function Int_Plus (Left, Right : Integer) return Integer renames "+";
```

16
```
function Rouge return Color renames Red;  -- see 3.5.1
function Rot   return Color renames Red;
function Rosso return Color renames Rouge;
```

17
```
function Next(X : Color) return Color renames Color'Succ; -- see 3.5.1
```

18 *Example of a subprogram renaming declaration with new parameter names:*

19
```
function "*" (X,Y : Vector) return Real renames Dot_Product; -- see 6.1
```

Example of a subprogram renaming declaration with a new default expression: 20

```
function Minimum(L : Link := Head) return Cell renames Min_Cell; -- see 6.1
```
21

Extensions to Ada 95

{*8652/0028*} {*AI95-00145-01*} **Corrigendum:** Allowed a renaming-as-body to be just mode conformant with the specification if the subprogram is not yet frozen. 21.a/2

{*AI95-00218-03*} Overriding_indicator (see 8.3.1) is optionally added to subprogram renamings. 21.b/2

Wording Changes from Ada 95

{*8652/0014*} {*AI95-00064-01*} **Corrigendum:** Described the semantics of renaming-as-body, so that the location of elaboration checks is clear. 21.c/2

{*8652/0027*} {*AI95-00135-01*} **Corrigendum:** Clarified that circular renaming-as-body is illegal (if it can be detected in time) or a bounded error. 21.d/2

{*AI95-00228-01*} **Amendment Correction:** Clarified that renaming a shall-be-overridden subprogram is illegal, as well as renaming-as-body an abstract subprogram. 21.e/2

{*AI95-00423-01*} Added matching rules for null_exclusions. 21.f/2

Inconsistencies With Ada 2005

{*AI05-0123-1*} Renaming of user-defined untagged record equality is now defined to call the overridden body so long as the overriding occurred before the renames. This could change the body called in unusual cases; the change is necessary to preserve the principle that the body called for an explicit call to "=" (via a renames in this case) is the same as the one inherited for a derived type and used in generics. Note that any renamings before the overriding will be unchanged. Any differences caused by the change will be rare and most likely will fix a bug. 21.f.1/3

Extensions to Ada 2005

{*AI05-0183-1*} An optional aspect_specification can be used in a subprogram_renaming_declaration. This is described in 13.1.1. 21.g/3

Incompatibilities With Ada 2012

{*AI12-0204-1*} **Correction:** Added a rule to ensure that the prefix of a renaming of a prefixed view continues to exist during the life of a renames. If the prefix is a subcomponent that depends on discriminants, Ada 2005 and 2012 would have allowed the prefix while Ada 2022 would not. Without this change, explicit forms (renaming the object and then using that in calls) would be safer than the renaming; that's inconsistent with other kinds of renaming. 21.h/5

Wording Changes from Ada 2012

{*AI12-0287-1*} **Correction:** Added wording to ensure that the object subject to a Legality Rule can be determined at compile-time. The alternative being nonsense, we treat this as a wording change. 21.i/5

8.5.5 Generic Renaming Declarations

[A generic_renaming_declaration is used to rename a generic unit.] 1

Syntax

{*AI05-0183-1*} generic_renaming_declaration ::= 2/3

generic package defining_program_unit_name **renames** *generic_package_*name
[aspect_specification];
| **generic procedure** defining_program_unit_name **renames** *generic_procedure_*name
[aspect_specification];
| **generic function** defining_program_unit_name **renames** *generic_function_*name
[aspect_specification];

Legality Rules

The renamed entity shall be a generic unit of the corresponding kind. 3

Static Semantics

4 A generic_renaming_declaration declares a new view [of the renamed generic unit].

5/5 NOTE {*AI12-0440-1*} Although the properties of the new view are the same as those of the renamed view, the place where the generic_renaming_declaration occurs can affect the legality of subsequent renamings and instantiations that denote the generic_renaming_declaration, in particular if the renamed generic unit is a library unit (see 10.1.1).

Examples

6 *Example of renaming a generic unit:*

7
```
generic package Enum_IO renames Ada.Text_IO.Enumeration_IO;   -- see A.10.10
```

Extensions to Ada 83

7.a Renaming of generic units is new to Ada 95. It is particularly important for renaming child library units that are generic units. For example, it might be used to rename Numerics.Generic_Elementary_Functions as simply Generic_Elementary_Functions, to match the name for the corresponding Ada-83-based package.

Wording Changes from Ada 83

7.b The information in RM83-8.6, "The Package Standard", has been updated for the child unit feature, and moved to Annex A, except for the definition of "predefined type", which has been moved to 3.2.1.

Extensions to Ada 2005

7.c/3 {*AI05-0183-1*} An optional aspect_specification can be used in a generic_renaming_declaration. This is described in 13.1.1.

8.6 The Context of Overload Resolution

1/3 {*AI05-0299-1*} [Because declarations can be overloaded, it is possible for an occurrence of a usage name to have more than one possible interpretation; in most cases, ambiguity is disallowed. This subclause describes how the possible interpretations resolve to the actual interpretation.

2 Certain rules of the language (the Name Resolution Rules) are considered "overloading rules". If a possible interpretation violates an overloading rule, it is assumed not to be the intended interpretation; some other possible interpretation is assumed to be the actual interpretation. On the other hand, violations of nonoverloading rules do not affect which interpretation is chosen; instead, they cause the construct to be illegal. To be legal, there usually has to be exactly one acceptable interpretation of a construct that is a "complete context", not counting any nested complete contexts.

3 The syntax rules of the language and the visibility rules given in 8.3 determine the possible interpretations. Most type checking rules (rules that require a particular type, or a particular class of types, for example) are overloading rules. Various rules for the matching of formal and actual parameters are overloading rules.]

Language Design Principles

3.a The type resolution rules are intended to minimize the need for implicit declarations and preference rules associated with implicit conversion and dispatching operations.

Name Resolution Rules

4 [Overload resolution is applied separately to each *complete context*, not counting inner complete contexts.] Each of the following constructs is a *complete context*:

5 • A context_item.

6 • A declarative_item or declaration.

6.a **Ramification:** A loop_parameter_specification is a declaration, and hence a complete context.

- A statement. 7
- A pragma_argument_association. 8

  Reason: We would make it the whole pragma, except that certain pragma arguments are allowed to be ambiguous, and ambiguity applies to a complete context. 8.a
- {*AI12-0040-1*} The *selecting_*expression of a case_statement or case_expression. 9/4

  Ramification: This means that the expression is resolved without looking at the choices. 9.a
- {*AI12-0125-3*} The *variable_*name of an assignment_statement *A*, if the expression of *A* contains one or more target_names. 9.1/5

An (overall) *interpretation* of a complete context embodies its meaning, and includes the following information about the constituents of the complete context, not including constituents of inner complete contexts: 10

- for each constituent of the complete context, to which syntactic categories it belongs, and by which syntax rules; and 11

  Ramification: Syntactic categor*ies* is plural here, because there are lots of trivial productions — an expression might also be all of the following, in this order: identifier, name, primary, factor, term, simple_expression, and relation. Basically, we're trying to capture all the information in the parse tree here, without using compiler-writer's jargon like "parse tree". 11.a
- for each usage name, which declaration it denotes (and, therefore, which view and which entity it denotes); and 12

  Ramification: {*AI95-00382-01*} In most cases, a usage name denotes the view declared by the denoted declaration. However, in certain cases, a usage name that denotes a declaration and appears inside the declarative region of that same declaration, denotes the current instance of the declaration. For example, within a task_body other than in an access_definition, a usage name that denotes the task_type_declaration denotes the object containing the currently executing task, and not the task type declared by the declaration. 12.a/2
- for a complete context that is a declarative_item, whether or not it is a completion of a declaration, and (if so) which declaration it completes. 13

  Ramification: Unfortunately, we are not confident that the above list is complete. We'll have to live with that. 13.a

  To be honest: For "possible" interpretations, the above information is tentative. 13.b

  Discussion: A possible interpretation (an *input* to overload resolution) contains information about what a usage name *might* denote, but what it actually *does* denote requires overload resolution to determine. Hence the term "tentative" is needed for possible interpretations; otherwise, the definition would be circular. 13.c

A *possible interpretation* is one that obeys the syntax rules and the visibility rules. An *acceptable interpretation* is a possible interpretation that obeys the *overloading rules*[, that is, those rules that specify an expected type or expected profile, or specify how a construct shall *resolve* or be *interpreted*.] 14

  To be honest: One rule that falls into this category, but does not use the above-mentioned magic words, is the rule about numbers of parameter associations in a call (see 6.4). 14.a

  Ramification: The Name Resolution Rules are the ones that appear under the Name Resolution Rules heading. Some Syntax Rules are written in English, instead of BNF. No rule is a Syntax Rule or Name Resolution Rule unless it appears under the appropriate heading. 14.b

The *interpretation* of a constituent of a complete context is determined from the overall interpretation of the complete context as a whole. [Thus, for example, "interpreted as a function_call", means that the construct's interpretation says that it belongs to the syntactic category function_call.] 15

[Each occurrence of] a usage name *denotes* the declaration determined by its interpretation. It also denotes the view declared by its denoted declaration, except in the following cases: 16

  Ramification: As explained below, a pragma argument is allowed to be ambiguous, so it can denote several declarations, and all of the views declared by those declarations. 16.a

- {*AI95-00382-01*} {*AI05-0287-1*} If a usage name appears within the declarative region of a type_declaration and denotes that same type_declaration, then it denotes the *current instance* of 17/3

the type (rather than the type itself); the current instance of a type is the object or value of the type that is associated with the execution that evaluates the usage name. Similarly, if a usage name appears within the declarative region of a subtype_declaration and denotes that same subtype_declaration, then it denotes the current instance of the subtype. These rules do not apply if the usage name appears within the subtype_mark of an access_definition for an access-to-object type, or within the subtype of a parameter or result of an access-to-subprogram type.

17.a/2 **Reason:** {*AI95-00382-01*} This is needed, for example, for references to the Access attribute from within the type_declaration. Also, within a task_body or protected_body, we need to be able to denote the current task or protected object. (For a single_task_declaration or single_protected_declaration, the rule about current instances is not needed.) We exclude anonymous access types so that they can be used to create self-referencing types in the natural manner (otherwise such types would be illegal).

17.b/2 **Discussion:** {*AI95-00382-01*} The phrase "within the subtype_mark" in the "this rule does not apply" part is intended to cover a case like **access** T'Class appearing within the declarative region of T: here T denotes the type, not the current instance.

17.1/5 {*AI12-0068-1*} {*AI12-0324-1*} {*AI12-0427-1*} Within an aspect_specification for a type or subtype, the current instance represents a value of the type; it is not an object. Unless otherwise specified, the nominal subtype of this value is given by the subtype itself (the first subtype in the case of a type_declaration), prior to applying any predicate specified directly on the type or subtype. If the type or subtype is by-reference, the associated object of the value is the object associated (see 6.2) with the evaluation of the usage name.

17.c/4 **Ramification:** For the purposes of Legality Rules, the current instance acts as a value within an aspect_specification. It might really be an object (and has to be for a by-reference type), but that isn't discoverable by direct use of the name of the current instance.

18 - If a usage name appears within the declarative region of a generic_declaration (but not within its generic_formal_part) and it denotes that same generic_declaration, then it denotes the *current instance* of the generic unit (rather than the generic unit itself). See also 12.3.

18.a **To be honest:** The current instance of a generic unit is the instance created by whichever generic_instantiation is of interest at any given time.

18.b **Ramification:** Within a generic_formal_part, a name that denotes the generic_declaration denotes the generic unit, which implies that it is not overloadable.

19 A usage name that denotes a view also denotes the entity of that view.

19.a **Ramification:** Usually, a usage name denotes only one declaration, and therefore one view and one entity.

20/2 {*AI95-00231-01*} The *expected type* for a given expression, name, or other construct determines, according to the *type resolution rules* given below, the types considered for the construct during overload resolution. [The type resolution rules provide support for class-wide programming, universal literals, dispatching operations, and anonymous access types:]

20.a **Ramification:** Expected types are defined throughout the RM95. The most important definition is that, for a subprogram, the expected type for the actual parameter is the type of the formal parameter.

20.b The type resolution rules are trivial unless either the actual or expected type is universal, class-wide, or of an anonymous access type.

21 - If a construct is expected to be of any type in a class of types, or of the universal or class-wide type for a class, then the type of the construct shall resolve to a type in that class or to a universal type that covers the class.

21.a **Ramification:** This matching rule handles (among other things) cases like the Val attribute, which denotes a function that takes a parameter of type *universal_integer*.

21.b/1 The last part of the rule, "or to a universal type that covers the class" implies that if the expected type for an expression is *universal_fixed*, then an expression whose type is *universal_real* (such as a real literal) is OK.

22 - If the expected type for a construct is a specific type *T*, then the type of the construct shall resolve either to *T*, or:

22.a **Ramification:** This rule is *not* intended to create a preference for the specific type — such a preference would cause Beaujolais effects.

- to *T*'Class; or 23

  Ramification: This will only be legal as part of a call on a dispatching operation; see 3.9.2, “Dispatching Operations of Tagged Types”. Note that that rule is not a Name Resolution Rule. 23.a

- to a universal type that covers *T*; or 24
- {*AI95-00230-01*} {*AI95-00231-01*} {*AI95-00254-01*} {*AI95-00409-01*} when *T* is a specific anonymous access-to-object type (see 3.10) with designated type *D*, to an access-to-object type whose designated type is *D*'Class or is covered by *D*; or 25/2

  This paragraph was deleted.{*AI95-00409-01*} 25.a/2

  Ramification: The case where the actual is access-to-*D*'Class will only be legal as part of a call on a dispatching operation; see 3.9.2, “Dispatching Operations of Tagged Types”. Note that that rule is not a Name Resolution Rule. 25.b

- {*AI05-0149-1*} when *T* is a named general access-to-object type (see 3.10) with designated type *D*, to an anonymous access-to-object type whose designated type covers or is covered by *D*; or 25.1/3
- {*AI95-00254-01*} {*AI95-00409-01*} {*AI05-0239-1*} when *T* is an anonymous access-to-subprogram type (see 3.10), to an access-to-subprogram type whose designated profile is type conformant with that of *T*. 25.2/3

In certain contexts, [such as in a subprogram_renaming_declaration,] the Name Resolution Rules define an *expected profile* for a given name; in such cases, the name shall resolve to the name of a callable entity whose profile is type conformant with the expected profile. 26

Ramification: {*AI05-0239-1*} The parameter and result *sub*types are not used in overload resolution. Only type conformance of profiles is considered during overload resolution. Legality rules generally require at least mode conformance in addition, but those rules are not used in overload resolution. 26.a/3

Legality Rules

{*AI95-00332-01*} When a construct is one that requires that its expected type be a *single* type in a given class, the type of the construct shall be determinable solely from the context in which the construct appears, excluding the construct itself, but using the requirement that it be in the given class. Furthermore, the context shall not be one that expects any type in some class that contains types of the given class; in particular, the construct shall not be the operand of a type_conversion. 27/2

Ramification: {*AI95-00230-01*} {*AI12-0005-1*} For example, the expected type for a character literal is required to be a single character type. But the expected type for the operand of a type_conversion is any type. Therefore, a character literal is not allowed as the operand of a type_conversion. This is true even if there is only one character type in scope (which is never the case). The reason for these rules is so that the compiler will not have to search “everywhere” to see if there is exactly one type in a class in scope. 27.a/5

Discussion: {*AI95-00332-01*} The first sentence is carefully worded so that it only mentions “expected type” as part of identifying the interesting case, but doesn't require that the context actually provide such an expected type. This allows such constructs to be used inside of constructs that don't provide an expected type (like qualified expressions and renames). Otherwise, such constructs wouldn't allow aggregates, 'Access, and so on. 27.b/2

{*AI05-0102-1*} {*AI05-0149-1*} {*AI05-0299-1*} {*AI12-0039-1*} Other than for the *tested*_simple_expression of a membership test, if the expected type for a name or expression is not the same as the actual type of the name or expression, the actual type shall be convertible to the expected type (see 4.6); further, if the expected type is a named access-to-object type with designated type *D1* and the actual type is an anonymous access-to-object type with designated type *D2*, then *D1* shall cover *D2*, and the name or expression shall denote a view with an accessibility level for which the statically deeper relationship applies[; in particular it shall not denote an access parameter nor a stand-alone access object]. 27.1/4

Reason: This rule prevents an implicit conversion that would be illegal if it was an explicit conversion. For instance, this prevents assigning an access-to-constant value into a stand-alone anonymous access-to-variable object. It also covers convertibility of the designated type and accessibility checks. 27.c/3

27.d/3 The rule also minimizes cases of implicit conversions when the tag check or the accessibility check might fail. We word it this way because access discriminants should also be disallowed if their enclosing object is designated by an access parameter.

27.e/3 **Ramification:** This rule does not apply to expressions that don't have expected types (such as the operand of a qualified expression or the expression of a renames). We don't need a rule like this in those cases, as the type needs to be the same; there is no implicit conversion.

28 A complete context shall have at least one acceptable interpretation; if there is exactly one, then that one is chosen.

28.a **Ramification:** This, and the rule below about ambiguity, are the ones that suck in all the Syntax Rules and Name Resolution Rules as compile-time rules. Note that this and the ambiguity rule have to be Legality Rules.

29 There is a *preference* for the primitive operators (and ranges) of the root numeric types *root_integer* and *root_real*. In particular, if two acceptable interpretations of a constituent of a complete context differ only in that one is for a primitive operator (or range) of the type *root_integer* or *root_real*, and the other is not, the interpretation using the primitive operator (or range) of the root numeric type is *preferred*.

29.a **Reason:** The reason for this preference is so that expressions involving literals and named numbers can be unambiguous. For example, without the preference rule, the following would be ambiguous:

29.b/1
```
N : constant := 123;
if N > 100 then -- Preference for root_integer ">" operator.
    ...
end if;
```

29.1/3 {*AI05-0149-1*} Similarly, there is a preference for the equality operators of the *universal_access* type (see 4.5.2). If two acceptable interpretations of a constituent of a complete context differ only in that one is for an equality operator of the *universal_access* type, and the other is not, the interpretation using the equality operator of the *universal_access* type is preferred.

29.c/3 **Reason:** This preference is necessary because of implicit conversion from an anonymous access type to a named access type, which would allow the equality operator of any named access type to be used to compare anonymous access values (and that way lies madness).

30 For a complete context, if there is exactly one overall acceptable interpretation where each constituent's interpretation is the same as or preferred (in the above sense) over those in all other overall acceptable interpretations, then that one overall acceptable interpretation is chosen. Otherwise, the complete context is *ambiguous*.

31 A complete context other than a pragma_argument_association shall not be ambiguous.

32 A complete context that is a pragma_argument_association is allowed to be ambiguous (unless otherwise specified for the particular pragma), but only if every acceptable interpretation of the pragma argument is as a name that statically denotes a callable entity. Such a name denotes all of the declarations determined by its interpretations, and all of the views declared by these declarations.

32.a/3 **Ramification:** {*AI95-00224-01*} {*AI05-0229-1*} This applies to Inline, Suppress, Import, Export, and Convention pragmas. For example, it is OK to say "**pragma** Export(C, Entity_Name => P.Q);", even if there are two directly visible P's, and there are two Q's declared in the visible part of each P. In this case, P.Q denotes four different declarations. This rule also applies to certain pragmas defined in the Specialized Needs Annexes. It almost applies to Pure, Elaborate_Body, and Elaborate_All pragmas, but those can't have overloading for other reasons. Note that almost all of these pragmas are obsolescent (see J.10 and J.15), and a major reason is that this rule has proven to be too broad in practice (it is common to want to specify something on a single subprogram of an overloaded set, that can't be done easily with this rule). Aspect_specifications, which are given on individual declarations, are preferred in Ada 2012.

32.b Note that if a pragma argument denotes a *call* to a callable entity, rather than the entity itself, this exception does not apply, and ambiguity is disallowed.

32.c Note that we need to carefully define which pragma-related rules are Name Resolution Rules, so that, for example, a pragma Inline does not pick up subprograms declared in enclosing declarative regions, and therefore make itself illegal.

32.d We say "statically denotes" in the above rule in order to avoid having to worry about how many times the name is evaluated, in case it denotes more than one callable entity.

NOTE If a usage name has only one acceptable interpretation, then it denotes the corresponding entity. However, this does not mean that the usage name is necessarily legal since other requirements exist which are not considered for overload resolution; for example, the fact that an expression is static, whether an object is constant, mode and subtype conformance rules, freezing rules, order of elaboration, and so on. 33

Similarly, subtypes are not considered for overload resolution (the violation of a constraint does not make a program illegal but raises an exception during program execution). 34

Incompatibilities With Ada 83

The new preference rule for operators of root numeric types is upward incompatible, but only in cases that involved *Beaujolais* effects in Ada 83. Such cases are ambiguous in Ada 95. 34.a

Extensions to Ada 83

The rule that allows an expected type to match an actual expression of a universal type, in combination with the new preference rule for operators of root numeric types, subsumes the Ada 83 "implicit conversion" rules for universal types. 34.b

Wording Changes from Ada 83

In Ada 83, it is not clear what the “syntax rules” are. AI83-00157 states that a certain textual rule is a syntax rule, but it's still not clear how one tells in general which textual rules are syntax rules. We have solved the problem by stating exactly which rules are syntax rules — the ones that appear under the “Syntax” heading. 34.c

RM83 has a long list of the “forms” of rules that are to be used in overload resolution (in addition to the syntax rules). It is not clear exactly which rules fall under each form. We have solved the problem by explicitly marking all rules that are used in overload resolution. Thus, the list of kinds of rules is unnecessary. It is replaced with some introductory (intentionally vague) text explaining the basic idea of what sorts of rules are overloading rules. 34.d

{*AI05-0299-1*} It is not clear from RM83 what information is embodied in a “meaning” or an “interpretation”. “Meaning” and “interpretation” were intended to be synonymous; we now use the latter only in defining the rules about overload resolution. “Meaning” is used only informally. This subclause attempts to clarify what is meant by “interpretation”. 34.e/3

For example, RM83 does not make it clear that overload resolution is required in order to match subprogram_bodies with their corresponding declarations (and even to tell whether a given subprogram_body is the completion of a previous declaration). Clearly, the information needed to do this is part of the “interpretation” of a subprogram_body. The resolution of such things is defined in terms of the “expected profile” concept. Ada 95 has some new cases where expected profiles are needed — the resolution of P'Access, where P might denote a subprogram, is an example. 34.f

RM83-8.7(2) might seem to imply that an interpretation embodies information about what is denoted by each usage name, but not information about which syntactic category each construct belongs to. However, it seems necessary to include such information, since the Ada grammar is highly ambiguous. For example, X(Y) might be a function_call or an indexed_component, and no context-free/syntactic information can tell the difference. It seems like we should view X(Y) as being, for example, “interpreted as a function_call” (if that's what overload resolution decides it is). Note that there are examples where the denotation of each usage name does not imply the syntactic category. However, even if that were not true, it seems that intuitively, the interpretation includes that information. Here's an example: 34.g

```
type T;                                                                    34.h
type A is access T;
type T is array(Integer range 1..10) of A;
I : Integer := 3;
function F(X : Integer := 7) return A;
Y : A := F(I); -- Ambiguous? (We hope so.)
```

Consider the declaration of Y (a complete context). In the above example, overload resolution can easily determine the declaration, and therefore the entity, denoted by Y, A, F, and I. However, given all of that information, we still don't know whether F(I) is a function_call or an indexed_component whose prefix is a function_call. (In the latter case, it is equivalent to F(7).**all**(I).) 34.i/1

It seems clear that the declaration of Y ought to be considered ambiguous. We describe that by saying that there are two interpretations, one as a function_call, and one as an indexed_component. These interpretations are both acceptable to the overloading rules. Therefore, the complete context is ambiguous, and therefore illegal. 34.j

It is the intent that the Ada 95 preference rule for root numeric operators is more locally enforceable than that of RM83-4.6(15). It should also eliminate interpretation shifts due to the addition or removal of a use_clause (the so called *Beaujolais* effect). 34.k

{*AI95-00114-01*} RM83-8.7 seems to be missing some complete contexts, such as pragma_argument_associations, declarative_items that are not declarations or aspect_clauses, and context_items. We have added these, and also 34.l/2

replaced the “must be determinable” wording of RM83-5.4(3) with the notion that the expression of a case_statement is a complete context.

34.m Cases like the Val attribute are now handled using the normal type resolution rules, instead of having special cases that explicitly allow things like “any integer type”.

Incompatibilities With Ada 95

34.n/2 {*AI95-00409-01*} Ada 95 allowed name resolution to distinguish between anonymous access-to-variable and access-to-constant types. This is similar to distinguishing between subprograms with **in** and **in out** parameters, which is known to be bad. Thus, that part of the rule was dropped as we now have anonymous access-to-constant types, making this much more likely.

34.o/2
```
type Cacc is access constant Integer;
procedure Proc (Acc : access Integer) ...
procedure Proc (Acc : Cacc) ...
List : Cacc := ...;
Proc (List); -- OK in Ada 95, ambiguous in Ada 2005.
```

34.p/2 If there is any code like this (such code should be rare), it will be ambiguous in Ada 2005.

Extensions to Ada 95

34.q/2 {*AI95-00230-01*} {*AI95-00231-01*} {*AI95-00254-01*} Generalized the anonymous access resolution rules to support the new capabilities of anonymous access types (that is, access-to-subprogram and access-to-constant).

34.r/2 {*AI95-00382-01*} We now allow the creation of self-referencing types via anonymous access types. This is an extension in unusual cases involving task and protected types. For example:

34.s/2
```
task type T;
```

34.t/2
```
task body T is
   procedure P (X : access T) is -- Illegal in Ada 95, legal in Ada 2005
      ...
   end P;
begin
   ...
end T;
```

Wording Changes from Ada 95

34.u/2 {*AI95-00332-01*} Corrected the “single expected type” so that it works in contexts that don't have expected types (like object renames and qualified expressions). This fixes a hole in Ada 95 that appears to prohibit using aggregates, 'Access, character literals, string literals, and allocators in qualified expressions.

Incompatibilities With Ada 2005

34.v/3 {*AI05-0149-1*} Implicit conversion is now allowed from anonymous access-to-object types to general access-to-object types. Such conversions can make calls ambiguous. That can only happen when there are two visible subprograms with the same name and have profiles that differ only by a parameter that is of a named or anonymous access type, and the actual argument is of an anonymous access type. This should be rare, as many possible calls would be ambiguous even in Ada 2005 (including allocators and any actual of a named access type if the designated types are the same).

Extensions to Ada 2005

34.w/3 {*AI05-0149-1*} Implicit conversion is allowed from anonymous access-to-object types to general access-to-object types if the designated type is convertible and runtime checks are minimized. See also the incompatibilities section.

Wording Changes from Ada 2005

34.x/3 {*AI05-0102-1*} Added a requirement here that implicit conversions are convertible to the appropriate type. This rule was scattered about the Reference Manual, we moved a single generalized version here.

Inconsistencies With Ada 2012

34.y/4 {*AI12-0068-1*} **Corrigendum:** Added a rule to specify that the current instance of a type or subtype is a value within an aspect_specification. This could be inconsistent if a predicate or invariant uses the Constrained attribute on the current instance (it will always be False now, while it might have returned True in original Ada 2012). More likely, a usage of a current instance as a prefix of an attribute will become illegal (such as Size or Alignment). Any such code is very tricky. Moreover, as this is a new feature of Ada 2012, there are not that many predicates and invariants, and the ones that exist are very unlikely to be this tricky. Thus we do not believe that there will be any practical effect to this change, other than to explicitly allow common implementation strategies.

Wording Changes from Ada 2012

{*AI12-0040-1*} **Corrigendum:** Added wording to clarify that the *selecting*_expression of a case_expression is a complete context, just like that of a case_statement. Clearly, everyone expects these to work the same way. Moreover, since it would be a lot of extra work to treat case_expressions differently, it is quite unlikely that any compiler would implement the much more complicated resolution necessary (and we are not aware of any that did). Therefore, we didn't document this as a potential incompatibility. 34.z/4

{*AI12-0125-3*} Added a resolution rule for target_name (see 5.2.1). 34.aa/5

9 Tasks and Synchronization

{*AI05-0299-1*} {*AI12-0119-1*} {*AI12-0330-1*} The execution of an Ada program consists of the execution of one or more *tasks*. Each task represents a separable activity that proceeds independently and concurrently between the points where it *interacts* with other tasks. A single task, when within the context of a parallel construct, can represent multiple *logical threads of control* which can proceed in parallel; in other contexts, each task represents one logical thread of control. 1/5

To be honest: The execution of an Ada program consists of the execution of one or more partitions (see 10.2), each of which in turn consists of the execution of an environment task and zero or more subtasks. 1.a

Term entry: logical thread of control — activity within the execution of a program that can proceed in parallel with other activities of the same task, or of separate tasks 1.b/5

{*AI12-0119-1*} The various forms of task interaction are described in this clause, and include: 1.1/5

- the activation and termination of a task; 2
- a call on a protected subprogram of a *protected object*, providing exclusive read-write access, or concurrent read-only access to shared data; 3
- a call on an entry, either of another task, allowing for synchronous communication with that task, or of a protected object, allowing for asynchronous communication with one or more other tasks using that same protected object; 4
- a timed operation, including a simple delay statement, a timed entry call or accept, or a timed asynchronous select statement (see next item); 5
- an asynchronous transfer of control as part of an asynchronous select statement, where a task stops what it is doing and begins execution at a different point in response to the completion of an entry call or the expiration of a delay; 6
- an abort statement, allowing one task to cause the termination of another task. 7

In addition, tasks can communicate indirectly by reading and updating (unprotected) shared variables, presuming the access is properly synchronized through some other kind of task interaction. 8

Static Semantics

The properties of a task are defined by a corresponding task declaration and task_body, which together define a program unit called a *task unit*. 9

Dynamic Semantics

{*AI12-0119-1*} Over time, tasks proceed through various *states*. A task is initially *inactive*; upon activation, and prior to its *termination* it is either *blocked* (as part of some task interaction) or *ready* to run. While ready, a task competes for the available *execution resources* that it requires to run. In the context of a parallel construct, a single task can utilize multiple processing resources simultaneously. 10/5

Discussion: {*AI05-0229-1*} The means for selecting which of the ready tasks to run, given the currently available execution resources, is determined by the *task dispatching policy* in effect, which is generally implementation defined, but may be controlled by aspects, pragmas, and operations defined in the Real-Time Annex (see D.2 and D.5). 10.a/3

NOTE {*AI12-0119-1*} {*AI12-0440-1*} Concurrent task execution can be implemented on multicomputers, multiprocessors, or with interleaved execution on a single physical processor. On the other hand, whenever an implementation can determine that the required semantic effects can be achieved when parts of the execution of a single logical thread of control are performed by different physical processors acting in parallel, it can choose to perform them in this way. 11/5

Wording Changes from Ada 83

The introduction has been rewritten. 11.a

11.b We use the term "concurrent" rather than "parallel" when talking about logically independent execution of threads of control. The term "parallel" is reserved for referring to the situation where multiple physical processors run simultaneously.

Wording Changes from Ada 2012

11.c/5 {*AI12-0119-1*} Rewrote the introduction to emphasize that threads of control can be generated by parallel constructs as well as tasks.

9.1 Task Units and Task Objects

1 A task unit is declared by a *task declaration*, which has a corresponding task_body. A task declaration may be a task_type_declaration, in which case it declares a named task type; alternatively, it may be a single_task_declaration, in which case it defines an anonymous task type, as well as declaring a named task object of that type.

Syntax

2/3 {*AI95-00345-01*} {*AI05-0183-1*} task_type_declaration ::=
task type defining_identifier [known_discriminant_part]
[aspect_specification] [**is**
[**new** interface_list **with**]
task_definition];

3/5 {*AI95-00399-01*} {*AI05-0183-1*} {*AI12-0446-1*} single_task_declaration ::=
task defining_identifier
[aspect_specification] [**is**
[**new** interface_list **with**]
task_definition];

4 task_definition ::=
{task_item}
[**private**
{task_item}]
end [*task_*identifier]

5/1 {*8652/0009*} {*AI95-00137-01*} task_item ::= entry_declaration | aspect_clause

6/3 {*AI05-0267-1*} task_body ::=
task body defining_identifier
[aspect_specification] **is**
declarative_part
begin
handled_sequence_of_statements
end [*task_*identifier];

7 If a *task_*identifier appears at the end of a task_definition or task_body, it shall repeat the defining_identifier.

8.a/2 *This paragraph was deleted.*

Paragraph 8 was deleted.

Static Semantics

A task_definition defines a task type and its first subtype. The first list of task_items of a task_definition, together with the known_discriminant_part, if any, is called the visible part of the task unit. [The optional list of task_items after the reserved word **private** is called the private part of the task unit.] 9

Proof: Private part is defined in Clause 8. 9.a

{*8652/0029*} {*AI95-00116-01*} For a task declaration without a task_definition, a task_definition without task_items is assumed. 9.1/1

{*AI95-00345-01*} {*AI95-00397-01*} {*AI95-00399-01*} {*AI95-00419-01*} {*AI05-0042-1*} For a task declaration with an interface_list, the task type inherits user-defined primitive subprograms from each progenitor type (see 3.9.4), in the same way that a derived type inherits user-defined primitive subprograms from its progenitor types (see 3.4). If the first parameter of a primitive inherited subprogram is of the task type or an access parameter designating the task type, and there is an entry_declaration for a single entry with the same identifier within the task declaration, whose profile is type conformant with the prefixed view profile of the inherited subprogram, the inherited subprogram is said to be *implemented* by the conforming task entry using an implicitly declared nonabstract subprogram which has the same profile as the inherited subprogram and which overrides it. 9.2/3

Ramification: The inherited subprograms can only come from an interface given as part of the task declaration. 9.b/2

Reason: {*AI05-0042-1*} The part about the implicitly declared subprogram is needed so that a subprogram implemented by an entry is considered to be overridden for the purpose of the other rules of the language. Without it, it would for instance be illegal for an abstract subprogram to be implemented by an entry, because the abstract subprogram would not be overridden. The Legality Rules below ensure that there is no conflict between the implicit overriding subprogram and a user-defined overriding subprogram. 9.b.1/3

Legality Rules

{*AI95-00345-01*} A task declaration requires a completion[, which shall be a task_body,] and every task_body shall be the completion of some task declaration. 9.3/2

To be honest: {*AI05-0229-1*} If the implementation supports it, the task body can be imported (using aspect Import, see B.1), in which case no explicit task_body is allowed. 9.c/3

{*AI95-00345-01*} {*AI95-00399-01*} [Each *interface*_subtype_mark of an interface_list appearing within a task declaration shall denote a limited interface type that is not a protected interface.] 9.4/2

Proof: 3.9.4 requires that an interface_list only name interface types, and limits the descendants of the various kinds of interface types. Only a limited, task, or synchronized interface can have a task type descendant. Nonlimited or protected interfaces are not allowed, as they offer operations that a task does not have. 9.d/2

{*AI95-00397-01*} {*AI05-0090-1*} The prefixed view profile of an explicitly declared primitive subprogram of a tagged task type shall not be type conformant with any entry of the task type, if the subprogram has the same defining name as the entry and the first parameter of the subprogram is of the task type or is an access parameter designating the task type. 9.5/3

Reason: This prevents the existence of two operations with the same name and profile which could be called with a prefixed view. If the operation was inherited, this would be illegal by the following rules; this rule puts inherited and noninherited routines on the same footing. Note that this only applies to tagged task types (that is, those with an interface in their declaration); we do that as there is no problem with prefixed view calls of primitive operations for "normal" task types, and having this rule apply to all tasks would be incompatible with Ada 95. 9.e/2

{*AI95-00345-01*} {*AI95-00399-01*} For each primitive subprogram inherited by the type declared by a task declaration, at most one of the following shall apply: 9.6/2

- {*AI95-00345-01*} the inherited subprogram is overridden with a primitive subprogram of the task type, in which case the overriding subprogram shall be subtype conformant with the inherited subprogram and not abstract; or 9.7/2

9.8/2 • {*AI95-00345-01*} {*AI95-00397-01*} the inherited subprogram is implemented by a single entry of the task type; in which case its prefixed view profile shall be subtype conformant with that of the task entry.

9.f/2 **Ramification:** An entry may implement two subprograms from the ancestors, one whose first parameter is of type *T* and one whose first parameter is of type **access** *T*. That doesn't cause implementation problems because "implemented by" (unlike "overridden') probably entails the creation of wrappers.

9.9/2 If neither applies, the inherited subprogram shall be a null procedure. In addition to the places where Legality Rules normally apply (see 12.3), these rules also apply in the private part of an instance of a generic unit.

9.g/2 **Reason:** Each inherited subprogram can only have a single implementation (either from overriding a subprogram or implementing an entry), and must have an implementation unless the subprogram is a null procedure.

Dynamic Semantics

10 [The elaboration of a task declaration elaborates the task_definition. The elaboration of a single_task_-declaration also creates an object of an (anonymous) task type.]

10.a **Proof:** This is redundant with the general rules for the elaboration of a full_type_declaration and an object_declaration.

11 [The elaboration of a task_definition creates the task type and its first subtype;] it also includes the elaboration of the entry_declarations in the given order.

12/1 {*8652/0009*} {*AI95-00137-01*} As part of the initialization of a task object, any aspect_clauses and any per-object constraints associated with entry_declarations of the corresponding task_definition are elaborated in the given order.

12.a/1 **Reason:** The only aspect_clauses defined for task entries are ones that specify the Address of an entry, as part of defining an interrupt entry. These clearly need to be elaborated per-object, not per-type. Normally the address will be a function of a discriminant, if such an Address clause is in a task type rather than a single task declaration, though it could rely on a parameterless function that allocates sequential interrupt vectors.

12.b We do not mention representation pragmas, since each pragma may have its own elaboration rules.

13 The elaboration of a task_body has no effect other than to establish that tasks of the type can from then on be activated without failing the Elaboration_Check.

14 [The execution of a task_body is invoked by the activation of a task of the corresponding type (see 9.2).]

15 The content of a task object of a given task type includes:

16 • The values of the discriminants of the task object, if any;

17 • An entry queue for each entry of the task object;

17.a **Ramification:** "For each entry" implies one queue for each single entry, plus one for each entry of each entry family.

18 • A representation of the state of the associated task.

19/2 NOTE 1 {*AI95-00382-01*} Other than in an access_definition, the name of a task unit within the declaration or body of the task unit denotes the current instance of the unit (see 8.6), rather than the first subtype of the corresponding task type (and thus the name cannot be used as a subtype_mark).

19.a/2 **Discussion:** {*AI95-00382-01*} It can be used as a subtype_mark in an anonymous access type. In addition, it is possible to refer to some other subtype of the task type within its body, presuming such a subtype has been declared between the task_type_declaration and the task_body.

20 NOTE 2 The notation of a selected_component can be used to denote a discriminant of a task (see 4.1.3). Within a task unit, the name of a discriminant of the task type denotes the corresponding discriminant of the current instance of the unit.

21/5 NOTE 3 {*AI95-00287-01*} {*AI12-0442-1*} A task type is a limited type (see 7.5), and hence precludes use of assignment_statements and predefined equality operators. If a programmer wants to write an application that stores and exchanges task identities, they can do so by defining an access type designating the corresponding task objects and by using access values for identification purposes. Assignment is available for such an access type as for any access type.

Alternatively, if the implementation supports the Systems Programming Annex, the Identity attribute can be used for task identification (see C.7.1).

Examples

Examples of declarations of task types: 22

```
task type Server is                                                                    23
   entry Next_Work_Item(WI : in Work_Item);
   entry Shut_Down;
end Server;

{AI95-00433-01} task type Keyboard_Driver(ID : Keyboard_ID := New_ID) is              24/2
      new Serial_Device with  -- see 3.9.4
   entry Read (C : out Character);
   entry Write(C : in  Character);
end Keyboard_Driver;
```

Examples of declarations of single tasks: 25

```
task Controller is                                                                     26
   entry Request(Level)(D : Item);  -- a family of entries
end Controller;

task Parser is                                                                         27
   entry Next_Lexeme(L : in  Lexical_Element);
   entry Next_Action(A : out Parser_Action);
end;

task User;  -- has no entries                                                          28
```

Examples of task objects: 29

```
Agent    : Server;                                                                     30
Teletype : Keyboard_Driver(TTY_ID);
Pool     : array(1 .. 10) of Keyboard_Driver;
```

Example of access type designating task objects: 31

```
type Keyboard is access Keyboard_Driver;                                               32
Terminal : Keyboard := new Keyboard_Driver(Term_ID);
```

Extensions to Ada 83

The syntax rules for task declarations are modified to allow a known_discriminant_part, and to allow a private part. They are also modified to allow entry_declarations and aspect_clauses to be mixed. 32.a/1

Wording Changes from Ada 83

The syntax rules for tasks have been split up according to task types and single tasks. In particular: The syntax rules for task_declaration and task_specification are removed. The syntax rules for task_type_declaration, single_task_declaration, task_definition and task_item are new. 32.b

The syntax rule for task_body now uses the nonterminal handled_sequence_of_statements. 32.c

The declarative_part of a task_body is now required; that doesn't make any real difference, because a declarative_part can be empty. 32.d

Extensions to Ada 95

{*AI95-00345-01*} {*AI95-00397-01*} {*AI95-00399-01*} {*AI95-00419-01*} Task types and single tasks can be derived from one or more interfaces. Entries of the task type can implement the primitive operations of an interface. Overriding_indicators can be used to specify whether or not an entry implements a primitive operation. 32.e/2

Wording Changes from Ada 95

{*8652/0029*} {*AI95-00116-01*} **Corrigendum:** Clarified that a task type has an implicit empty task_definition if none is given. 32.f/2

{*8652/0009*} {*AI95-00137-01*} **Corrigendum:** Changed representation clauses to aspect clauses to reflect that they are used for more than just representation. 32.g/2

32.h/2 {*AI95-00287-01*} Revised the note on operations of task types to reflect that limited types do have an assignment operation, but not copying (assignment_statements).

32.i/2 {*AI95-00382-01*} Revised the note on use of the name of a task type within itself to reflect the exception for anonymous access types.

Extensions to Ada 2005

32.j/3 {*AI05-0183-1*} {*AI05-0267-1*} An optional aspect_specification can be used in a task_type_declaration, a single_task_declaration, and a task_body. This is described in 13.1.1.

Wording Changes from Ada 2005

32.k/3 {*AI05-0042-1*} **Correction:** Clarified that an inherited procedure of a progenitor is overridden when it is implemented by an entry.

32.l/3 {*AI05-0090-1*} **Correction:** Added the missing defining name in the no conflicting primitive operation rule.

9.2 Task Execution - Task Activation

Dynamic Semantics

1 The execution of a task of a given task type consists of the execution of the corresponding task_body. The initial part of this execution is called the *activation* of the task; it consists of the elaboration of the declarative_part of the task_body. Should an exception be propagated by the elaboration of its declarative_part, the activation of the task is defined to have *failed*, and it becomes a completed task.

2/2 {*AI95-00416-01*} A task object (which represents one task) can be a part of a stand-alone object, of an object created by an allocator, or of an anonymous object of a limited type, or a coextension of one of these. All tasks that are part or coextensions of any of the stand-alone objects created by the elaboration of object_declarations (or generic_associations of formal objects of mode **in**) of a single declarative region are activated together. All tasks that are part or coextensions of a single object that is not a stand-alone object are activated together.

2.a **Discussion:** The initialization of an object_declaration or allocator can indirectly include the creation of other objects that contain tasks. For example, the default expression for a subcomponent of an object created by an allocator might call a function that evaluates a completely different allocator. Tasks created by the two allocators are *not* activated together.

3/2 {*AI95-00416-01*} For the tasks of a given declarative region, the activations are initiated within the context of the handled_sequence_of_statements (and its associated exception_handlers if any — see 11.2), just prior to executing the statements of the handled_sequence_of_statements. [For a package without an explicit body or an explicit handled_sequence_of_statements, an implicit body or an implicit null_statement is assumed, as defined in 7.2.]

3.a **Ramification:** If Tasking_Error is raised, it can be handled by handlers of the handled_sequence_of_statements.

4/2 {*AI95-00416-01*} For tasks that are part or coextensions of a single object that is not a stand-alone object, activations are initiated after completing any initialization of the outermost object enclosing these tasks, prior to performing any other operation on the outermost object. In particular, for tasks that are part or coextensions of the object created by the evaluation of an allocator, the activations are initiated as the last step of evaluating the allocator, prior to returning the new access value. For tasks that are part or coextensions of an object that is the result of a function call, the activations are not initiated until after the function returns.

4.a/2 **Discussion:** {*AI95-00416-01*} The intent is that "temporary" objects with task parts (or coextensions) are treated similarly to an object created by an allocator. The "whole" object is initialized, and then all of the task parts (including the coextensions) are activated together. Each such "whole" object has its own task activation sequence, involving the activating task being suspended until all the new tasks complete their activation.

The task that created the new tasks and initiated their activations (the *activator*) is blocked until all of these activations complete (successfully or not). Once all of these activations are complete, if the activation of any of the tasks has failed [(due to the propagation of an exception)], Tasking_Error is raised in the activator, at the place at which it initiated the activations. Otherwise, the activator proceeds with its execution normally. Any tasks that are aborted prior to completing their activation are ignored when determining whether to raise Tasking_Error. 5

Ramification: Note that a task created by an allocator does not necessarily depend on its activator; in such a case the activator's termination can precede the termination of the newly created task. 5.a

Discussion: Tasking_Error is raised only once, even if two or more of the tasks being activated fail their activation. 5.b

To be honest: {*AI95-00265-01*} The pragma Partition_Elaboration_Policy (see H.6) can be used to defer task activation to a later point, thus changing many of these rules. 5.c/2

{*AI05-0045-1*} If the master that directly encloses the point where the activation of a task *T* would be initiated, completes before the activation of *T* is initiated, *T* becomes terminated and is never activated. Furthermore, if a return statement is left such that the return object is not returned to the caller, any task that was created as a part of the return object or one of its coextensions immediately becomes terminated and is never activated. 6/3

Ramification: {*AI05-0045-1*} The first case can only happen if the activation point of T is not reached due to an exception being raised or a task or statement being aborted. Note that this is exclusive; if the master completes normally and starts finalization, we're already past the activation point. 6.a/3

{*AI05-0045-1*} The second case can happen with an exception being raised in a return statement, by an exit or goto from an extended_return_statement, or by a return statement being aborted. Any tasks created for the return object of such a return statement are never activated. 6.b/3

NOTE 1 An entry of a task can be called before the task has been activated. 7

NOTE 2 {*AI12-0442-1*} If several tasks are activated together, the execution of any of these tasks can proceed without waiting until the end of the activation of the other tasks. 8/5

NOTE 3 A task can become completed during its activation either because of an exception or because it is aborted (see 9.8). 9

Examples

Example of task activation: 10

```
procedure P is                                                    11
   A, B : Server;     -- elaborate the task objects A, B
   C    : Server;     -- elaborate the task object C
begin
   -- the tasks A, B, C are activated together before the first statement
   ...
end;
```

Wording Changes from Ada 83

We have replaced the term *suspended* with *blocked*, since we didn't want to consider a task blocked when it was simply competing for execution resources. "Suspended" is sometimes used more generally to refer to tasks that are not actually running on some processor, due to the lack of resources. 11.a

{*AI05-0299-1*} This subclause has been rewritten in an attempt to improve presentation. 11.b/3

Wording Changes from Ada 95

{*AI95-00416-01*} Adjusted the wording for activating tasks to handle the case of anonymous function return objects. This is critical; we don't want to be waiting for the tasks in a return object when we exit the function normally. 11.c/2

Wording Changes from Ada 2005

{*AI05-0045-1*} **Correction:** Corrected the wording that handles tasks that are never activated to ensure that no lookahead is implied and to make it clear that tasks created by return statements that never return are never activated. 11.d/3

9.3 Task Dependence - Termination of Tasks

Dynamic Semantics

1 Each task (other than an environment task — see 10.2) *depends* on one or more masters (see 7.6.1), as follows:

2/4 • {*AI12-0070-1*} If the task is created by the evaluation of an allocator for a given named access type, it depends on each master that includes the elaboration of the declaration of the ultimate ancestor of the given access type.

3 • If the task is created by the elaboration of an object_declaration, it depends on each master that includes this elaboration.

3.1/2 • {*AI95-00416-01*} Otherwise, the task depends on the master of the outermost object of which it is a part (as determined by the accessibility level of that object — see 3.10.2 and 7.6.1), as well as on any master whose execution includes that of the master of the outermost object.

3.a/2 **Ramification:** {*AI95-00416-01*} The master of a task created by a return statement changes when the accessibility of the return object changes. Note that its activation happens, if at all, only after the function returns and all accessibility level changes have occurred.

4 Furthermore, if a task depends on a given master, it is defined to depend on the task that executes the master, and (recursively) on any master of that task.

4.a **Discussion:** Don't confuse these kinds of dependences with the dependences among compilation units defined in 10.1.1, “Compilation Units - Library Units”.

5 A task is said to be *completed* when the execution of its corresponding task_body is completed. A task is said to be *terminated* when any finalization of the task_body has been performed (see 7.6.1). [The first step of finalizing a master (including a task_body) is to wait for the termination of any tasks dependent on the master.] The task executing the master is blocked until all the dependents have terminated. [Any remaining finalization is then performed and the master is left.]

6/1 Completion of a task (and the corresponding task_body) can occur when the task is blocked at a select_statement with an open terminate_alternative (see 9.7.1); the open terminate_alternative is selected if and only if the following conditions are satisfied:

7/2 • {*AI95-00415-01*} The task depends on some completed master; and

8 • Each task that depends on the master considered is either already terminated or similarly blocked at a select_statement with an open terminate_alternative.

9 When both conditions are satisfied, the task considered becomes completed, together with all tasks that depend on the master considered that are not yet completed.

9.a **Ramification:** Any required finalization is performed after the selection of terminate_alternatives. The tasks are not callable during the finalization. In some ways it is as though they were aborted.

10 NOTE 1 The full view of a limited private type can be a task type, or can have subcomponents of a task type. Creation of an object of such a type creates dependences according to the full type.

11 NOTE 2 An object_renaming_declaration defines a new view of an existing entity and hence creates no further dependence.

12/5 NOTE 3 {*AI12-0440-1*} The rules given for the collective completion of a group of tasks all blocked on select_statements with open terminate_alternatives ensure that the collective completion can occur only when there are no remaining active tasks that can call one of the tasks being collectively completed.

13 NOTE 4 If two or more tasks are blocked on select_statements with open terminate_alternatives, and become completed collectively, their finalization actions proceed concurrently.

14 NOTE 5 The completion of a task can occur due to any of the following:

15 • the raising of an exception during the elaboration of the declarative_part of the corresponding task_body;

- the completion of the handled_sequence_of_statements of the corresponding task_body; 16
- the selection of an open terminate_alternative of a select_statement in the corresponding task_body; 17
- the abort of the task. 18

Examples

Example of task dependence: 19

```
declare                                                                    20
   type Global is access Server;         -- see 9.1
   A, B : Server;
   G    : Global;
begin
   -- activation of A and B
   declare
      type Local is access Server;
      X : Global := new Server;  -- activation of X.all
      L : Local  := new Server;  -- activation of L.all
      C : Server;
   begin
      -- activation of C
      G := X;  -- both G and X designate the same task object
      ...
   end;  -- await termination of C and L.all (but not X.all)
   ...
end;  -- await termination of A, B, and G.all
```

Wording Changes from Ada 83

We have revised the wording to be consistent with the definition of master now given in 7.6.1, “Completion and Finalization”. 20.a

Tasks that used to depend on library packages in Ada 83, now depend on the (implicit) task_body of the environment task (see 10.2). Therefore, the environment task has to wait for them before performing library level finalization and terminating the partition. In Ada 83 the requirement to wait for tasks that depended on library packages was not as clear. 20.b

What was "collective termination" is now "collective completion" resulting from selecting terminate_alternatives. This is because finalization still occurs for such tasks, and this happens after selecting the terminate_alternative, but before termination. 20.c

Wording Changes from Ada 95

{*AI95-00416-01*} Added missing wording that explained the master of tasks that are neither object_declarations nor allocators, such as function returns. 20.d/2

Wording Changes from Ada 2012

{*AI12-0070-1*} **Corrigendum:** Ensured that the master of tasks that are not allocators of named access types is correctly determined. (Ignoring the accessibility rules of 3.10.2 could not be intended.) 20.e/4

9.4 Protected Units and Protected Objects

A *protected object* provides coordinated access to shared data, through calls on its visible *protected operations*, which can be *protected subprograms* or *protected entries*. A *protected unit* is declared by a *protected declaration*, which has a corresponding protected_body. A protected declaration may be a protected_type_declaration, in which case it declares a named protected type; alternatively, it may be a single_protected_declaration, in which case it defines an anonymous protected type, as well as declaring a named protected object of that type. 1

Syntax

{*AI95-00345-01*} {*AI05-0183-1*} protected_type_declaration ::= 2/3
protected type defining_identifier [known_discriminant_part]

[aspect_specification] **is**
[**new** interface_list **with**]
protected_definition;

3/3 {*AI95-00399-01*} {*AI05-0183-1*} single_protected_declaration ::=
protected defining_identifier
[aspect_specification] **is**
[**new** interface_list **with**]
protected_definition;

4 protected_definition ::=
{ protected_operation_declaration }
[**private**
{ protected_element_declaration }]
end [*protected*_identifier]

5/1 {*8652/0009*} {*AI95-00137-01*} protected_operation_declaration ::= subprogram_declaration
| entry_declaration
| aspect_clause

6 protected_element_declaration ::= protected_operation_declaration
| component_declaration

6.a **Reason:** We allow the operations and components to be mixed because that's how other things work (for example, package declarations). We have relaxed the ordering rules for the items inside declarative_parts and task_definitions as well.

7/3 {*AI05-0267-1*} protected_body ::=
protected body defining_identifier
[aspect_specification] **is**
{ protected_operation_item }
end [*protected*_identifier];

8/4 {*8652/0009*} {*AI95-00137-01*} {*AI12-0147-1*} protected_operation_item ::=
subprogram_declaration
| subprogram_body
| null_procedure_declaration
| expression_function_declaration
| entry_body
| aspect_clause

9 If a *protected*_identifier appears at the end of a protected_definition or protected_body, it shall repeat the defining_identifier.

10.a/2 *This paragraph was deleted.*

Paragraph 10 was deleted.

Static Semantics

11/2 {*AI95-00345-01*} {*AI95-00401-01*} A protected_definition defines a protected type and its first subtype. The list of protected_operation_declarations of a protected_definition, together with the known_discriminant_part, if any, is called the visible part of the protected unit. [The optional list of protected_element_declarations after the reserved word **private** is called the private part of the protected unit.]

11.a **Proof:** Private part is defined in Clause 8.

{*AI95-00345-01*} {*AI95-00397-01*} {*AI95-00399-01*} {*AI95-00419-01*} {*AI05-0042-1*} For a protected declaration with an interface_list, the protected type inherits user-defined primitive subprograms from each progenitor type (see 3.9.4), in the same way that a derived type inherits user-defined primitive subprograms from its progenitor types (see 3.4). If the first parameter of a primitive inherited subprogram is of the protected type or an access parameter designating the protected type, and there is a protected_operation_declaration for a protected subprogram or single entry with the same identifier within the protected declaration, whose profile is type conformant with the prefixed view profile of the inherited subprogram, the inherited subprogram is said to be *implemented* by the conforming protected subprogram or entry using an implicitly declared nonabstract subprogram which has the same profile as the inherited subprogram and which overrides it. 11.1/3

Ramification: The inherited subprograms can only come from an interface given as part of the protected declaration. 11.b/2

Reason: {*AI05-0042-1*} The part about the implicitly declared subprogram is needed so that a subprogram implemented by an entry or subprogram is considered to be overridden for the purpose of the other rules of the language. Without it, it would for instance be illegal for an abstract subprogram to be implemented by an entry, because the abstract subprogram would not be overridden. The Legality Rules below ensure that there is no conflict between the implicit overriding subprogram and a user-defined overriding subprogram. 11.b.1/3

Legality Rules

{*AI95-00345-01*} A protected declaration requires a completion[, which shall be a protected_body,] and every protected_body shall be the completion of some protected declaration. 11.2/2

To be honest: {*AI05-0229-1*} If the implementation supports it, the protected body can be imported (using aspect Import, see B.1), in which case no explicit protected_body is allowed. 11.c/3

{*AI95-00345-01*} {*AI95-00399-01*} [Each *interface*_subtype_mark of an interface_list appearing within a protected declaration shall denote a limited interface type that is not a task interface.] 11.3/2

Proof: 3.9.4 requires that an interface_list only name interface types, and limits the descendants of the various kinds of interface types. Only a limited, protected, or synchronized interface can have a protected type descendant. Nonlimited or task interfaces are not allowed, as they offer operations that a protected type does not have. 11.d/2

{*AI95-00397-01*} {*AI05-0042-1*} The prefixed view profile of an explicitly declared primitive subprogram of a tagged protected type shall not be type conformant with any protected operation of the protected type, if the subprogram has the same defining name as the protected operation and the first parameter of the subprogram is of the protected type or is an access parameter designating the protected type. 11.4/3

Reason: This prevents the existence of two operations with the same name and profile which could be called with a prefixed view. If the operation was inherited, this would be illegal by the following rules; this rule puts inherited and noninherited routines on the same footing. Note that this only applies to tagged protected types (that is, those with an interface in their declaration); we do that as there is no problem with prefixed view calls of primitive operations for "normal" protected types, and having this rule apply to all protected types would be incompatible with Ada 95. 11.e/2

{*AI95-00345-01*} {*AI95-00399-01*} For each primitive subprogram inherited by the type declared by a protected declaration, at most one of the following shall apply: 11.5/2

- {*AI95-00345-01*} the inherited subprogram is overridden with a primitive subprogram of the protected type, in which case the overriding subprogram shall be subtype conformant with the inherited subprogram and not abstract; or 11.6/2
- {*AI95-00345-01*} {*AI95-00397-01*} the inherited subprogram is implemented by a protected subprogram or single entry of the protected type, in which case its prefixed view profile shall be subtype conformant with that of the protected subprogram or entry. 11.7/2

If neither applies, the inherited subprogram shall be a null procedure. In addition to the places where Legality Rules normally apply (see 12.3), these rules also apply in the private part of an instance of a generic unit. 11.8/2

11.f/2 **Reason:** Each inherited subprogram can only have a single implementation (either from overriding a subprogram, implementing a subprogram, or implementing an entry), and must have an implementation unless the subprogram is a null procedure.

11.9/3 {*AI95-00345-01*} {*AI05-0291-1*} If an inherited subprogram is implemented by a protected procedure or an entry, then the first parameter of the inherited subprogram shall be of mode **out** or **in out**, or an access-to-variable parameter. If an inherited subprogram is implemented by a protected function, then the first parameter of the inherited subprogram shall be of mode **in**, but not an access-to-variable parameter.

11.g/3 **Reason:** For a protected procedure or entry, the protected object can be read or written (see 9.5.1). A subprogram that is implemented by a protected procedure or entry must have a profile which reflects that in order to avoid confusion. Similarly, a protected function has a parameter that is a constant, and the inherited routine should reflect that.

11.10/2 {*AI95-00397-01*} If a protected subprogram declaration has an overriding_indicator, then at the point of the declaration:

11.11/2 - if the overriding_indicator is **overriding**, then the subprogram shall implement an inherited subprogram;

11.12/2 - if the overriding_indicator is **not overriding**, then the subprogram shall not implement any inherited subprogram.

11.13/2 In addition to the places where Legality Rules normally apply (see 12.3), these rules also apply in the private part of an instance of a generic unit.

11.h/2 **Discussion:** These rules are subtly different than those for subprograms (see 8.3.1) because there cannot be "late" inheritance of primitives from interfaces. Hidden (that is, private) interfaces are prohibited explicitly (see 7.3), as are hidden primitive operations (as private operations of public abstract types are prohibited — see 3.9.3).

Dynamic Semantics

12 [The elaboration of a protected declaration elaborates the protected_definition. The elaboration of a single_protected_declaration also creates an object of an (anonymous) protected type.]

12.a **Proof:** This is redundant with the general rules for the elaboration of a full_type_declaration and an object_declaration.

13 [The elaboration of a protected_definition creates the protected type and its first subtype;] it also includes the elaboration of the component_declarations and protected_operation_declarations in the given order.

14 [As part of the initialization of a protected object, any per-object constraints (see 3.8) are elaborated.]

14.a **Discussion:** We do not mention pragmas since each pragma has its own elaboration rules.

15 The elaboration of a protected_body has no other effect than to establish that protected operations of the type can from then on be called without failing the Elaboration_Check.

16 The content of an object of a given protected type includes:

17 - The values of the components of the protected object, including (implicitly) an entry queue for each entry declared for the protected object;

17.a **Ramification:** "For each entry" implies one queue for each single entry, plus one for each entry of each entry family.

18 - A representation of the state of the execution resource *associated* with the protected object (one such resource is associated with each protected object).

19 [The execution resource associated with a protected object has to be acquired to read or update any components of the protected object; it can be acquired (as part of a protected action — see 9.5.1) either for concurrent read-only access, or for exclusive read-write access.]

20 As the first step of the *finalization* of a protected object, each call remaining on any entry queue of the object is removed from its queue and Program_Error is raised at the place of the corresponding entry_call_statement.

Reason: This is analogous to the raising of Tasking_Error in callers of a task that completes before accepting the calls. This situation can only occur due to a requeue (ignoring premature unchecked_deallocation), since any task that has accessibility to a protected object is awaited before finalizing the protected object. For example: 20.a

```
procedure Main is                                                    20.b
    task T is
        entry E;
    end T;

    task body T is                                                   20.c
        protected PO is
            entry Ee;
        end PO;

        protected body PO is                                         20.d
            entry Ee when False is
            begin
                null;
            end Ee;
        end PO;
    begin
        accept E do
            requeue PO.Ee;
        end E;
    end T;
begin
    T.E;
end Main;
```

{*AI05-0005-1*} The environment task is queued on PO.Ee when PO is finalized. 20.e/3

In a real example, a server task might park callers on a local protected object for some useful purpose, so we didn't want to disallow this case. 20.f

Bounded (Run-Time) Errors

{*AI95-00280-01*} It is a bounded error to call an entry or subprogram of a protected object after that object is finalized. If the error is detected, Program_Error is raised. Otherwise, the call proceeds normally, which may leave a task queued forever. 20.1/2

Reason: This is very similar to the finalization rule. It is a bounded error so that an implementation can avoid the overhead of the check if it can ensure that the call still will operate properly. Such an implementation cannot need to return resources (such as locks) to an executive that it needs to execute calls. 20.g/2

This case can happen (and has happened in production code) when a protected object is accessed from the Finalize routine of a type. For example: 20.h/2

```
with Ada.Finalization.Controlled;                                    20.i/2
package Window_Manager is
    ...
    type Root_Window is new Ada.Finalization.Controlled with private;
    type Any_Window is access all Root_Window;
    ...
private
    ...
    procedure Finalize (Object : in out Root_Window);
    ...
end Window_Manager;

package body Window_Manager is                                       20.j/2
   protected type Lock is
       entry Get_Lock;
       procedure Free_Lock;
   ...
   end Lock;

   Window_Lock : Lock;                                               20.k/2

   procedure Finalize (Object : in out Root_Window) is               20.l/2
   begin
       Window_Lock.Get_Lock;
       ...
       Window_Lock.Free_Lock;
   end Finalize;
   ...
   A_Window : Any_Window := new Root_Window;
end Window_Manager;
```

20.m/2 The environment task will call Window_Lock for the object allocated for A_Window when the collection for Any_Window is finalized, which will happen after the finalization of Window_Lock (because finalization of the package body will occur before that of the package specification).

21/2 NOTE 1 {*AI95-00382-01*} Within the declaration or body of a protected unit other than in an access_definition, the name of the protected unit denotes the current instance of the unit (see 8.6), rather than the first subtype of the corresponding protected type (and thus the name cannot be used as a subtype_mark).

21.a/2 **Discussion:** {*AI95-00382-01*} It can be used as a subtype_mark in an anonymous access type. In addition, it is possible to refer to some other subtype of the protected type within its body, presuming such a subtype has been declared between the protected_type_declaration and the protected_body.

22 NOTE 2 A selected_component can be used to denote a discriminant of a protected object (see 4.1.3). Within a protected unit, the name of a discriminant of the protected type denotes the corresponding discriminant of the current instance of the unit.

23/2 NOTE 3 {*AI95-00287-01*} A protected type is a limited type (see 7.5), and hence precludes use of assignment_statements and predefined equality operators.

24 NOTE 4 The bodies of the protected operations given in the protected_body define the actions that take place upon calls to the protected operations.

25 NOTE 5 The declarations in the private part are only visible within the private part and the body of the protected unit.

25.a **Reason:** Component_declarations are disallowed in a protected_body because, for efficiency, we wish to allow the compiler to determine the size of protected objects (when not dynamic); the compiler cannot necessarily see the body. Furthermore, the semantics of initialization of such objects would be problematic — we do not wish to give protected objects complex initialization semantics similar to task activation.

25.b The same applies to entry_declarations, since an entry involves an implicit component — the entry queue.

Examples

26 *Example of declaration of protected type and corresponding body:*

27
```
protected type Resource is
   entry Seize;
   procedure Release;
private
   Busy : Boolean := False;
end Resource;
```

28
```
protected body Resource is
   entry Seize when not Busy is
   begin
      Busy := True;
   end Seize;
```

29
```
   procedure Release is
   begin
      Busy := False;
   end Release;
end Resource;
```

30 *Example of a single protected declaration and corresponding body:*

31/5
```
{AI12-0430-1} protected Shared_Array is
   --  Index, Item, and Item_Array are global types
   function  Component    (N : in Index) return Item;
   procedure Set_Component(N : in Index; E : in  Item);
private
   Table : Item_Array(Index) := (others => Null_Item);
end Shared_Array;
```

32
```
protected body Shared_Array is
   function Component(N : in Index) return Item is
   begin
      return Table(N);
   end Component;
```

```
   procedure Set_Component(N : in Index; E : in Item) is
   begin
      Table(N) := E;
   end Set_Component;
end Shared_Array;
```

Examples of protected objects:

```
Control  : Resource;
Flags    : array(1 .. 100) of Resource;
```

Extensions to Ada 83

{*AI05-0299-1*} This entire subclause is new; protected units do not exist in Ada 83.

Extensions to Ada 95

{*AI95-00345-01*} {*AI95-00397-01*} {*AI95-00399-01*} {*AI95-00401-01*} {*AI95-00419-01*} Protected types and single protected objects can be derived from one or more interfaces. Operations declared in the protected type can implement the primitive operations of an interface. Overriding_indicators can be used to specify whether or not a protected operation implements a primitive operation.

Wording Changes from Ada 95

{*8652/0009*} {*AI95-00137-01*} **Corrigendum:** Changed representation clauses to aspect clauses to reflect that they are used for more than just representation.

{*AI95-00280-01*} Described what happens when an operation of a finalized protected object is called.

{*AI95-00287-01*} Revised the note on operations of protected types to reflect that limited types do have an assignment operation, but not copying (assignment_statements).

{*AI95-00382-01*} Revised the note on use of the name of a protected type within itself to reflect the exception for anonymous access types.

Incompatibilities With Ada 2005

{*AI05-0291-1*} **Correction:** When an inherited subprogram is implemented by a protected function, the first parameter has to be an **in** parameter, but not an access-to-variable type. Original Ada 2005 allowed access-to-variable parameters in this case; the parameter will need to be changed to access-to-constant with the addition of the **constant** keyword.

Extensions to Ada 2005

{*AI05-0183-1*} {*AI05-0267-1*} An optional aspect_specification can be used in a protected_type_declaration, a single_protected_declaration, and a protected_body. This is described in 13.1.1.

Wording Changes from Ada 2005

{*AI05-0042-1*} **Correction:** Clarified that an inherited subprogram of a progenitor is overridden when it is implemented by an entry or subprogram.

{*AI05-0090-1*} **Correction:** Added the missing defining name in the no conflicting primitive operation rule.

Extensions to Ada 2012

{*AI12-0147-1*} {*AI12-0005-1*} **Corrigendum:** Null procedures and expression functions are allowed in protected bodies. We consider this an omission, as there is no reason why the convenient shorthand notations should not be allowed in this context.

9.5 Intertask Communication

The primary means for intertask communication is provided by calls on entries and protected subprograms. Calls on protected subprograms allow coordinated access to shared data objects. Entry calls allow for blocking the caller until a given condition is satisfied (namely, that the corresponding entry is open — see 9.5.3), and then communicating data or control information directly with another task or indirectly via a shared protected object.

Static Semantics

2/3 {*AI05-0225-1*} {*AI05-0291-1*} When a name or prefix denotes an entry, protected subprogram, or a prefixed view of a primitive subprogram of a limited interface whose first parameter is a controlling parameter, the name or prefix determines a *target object*, as follows:

2.a/3 **To be honest:** {*AI05-0291-1*} This wording uses "denotes" to mean "denotes a view of an entity" (when the term is used in Legality Rules), and "denotes an entity" (when the term is used in Dynamic Semantics rules). It does not mean "view of a declaration", as that would not include renames (a renames is not an entry or protected subprogram).

3/3 - {*AI05-0291-1*} If it is a direct_name or expanded name that denotes the declaration (or body) of the operation, then the target object is implicitly specified to be the current instance of the task or protected unit immediately enclosing the operation; a call using such a name is defined to be an *internal call*;

4/3 - {*AI05-0291-1*} If it is a selected_component that is not an expanded name, then the target object is explicitly specified to be the object denoted by the prefix of the name; a call using such a name is defined to be an *external call*;

4.a **Discussion:** For example:

4.b
```
protected type Pt is
  procedure Op1;
  procedure Op2;
end Pt;
```

4.c
```
PO : Pt;
Other_Object : Some_Other_Protected_Type;
```

4.d
```
protected body Pt is
  procedure Op1 is begin ... end Op1;
```

4.e/5
```
{AI12-0005-1}   procedure Op2 is
  begin
    Op1; -- An internal call.
    Pt.Op1; -- Another internal call.
    PO.Op1; -- An external call. If the current instance is PO, then
            -- this is a bounded error (see 9.5.1).
    Other_Object.Some_Op; -- An external call.
  end Op2;
end Pt;
```

5/3 - {*AI05-0291-1*} If the name or prefix is a dereference (implicit or explicit) of an access-to-protected-subprogram value, then the target object is determined by the prefix of the Access attribute_reference that produced the access value originally; a call using such a name is defined to be an *external call*;

6 - If the name or prefix denotes a subprogram_renaming_declaration, then the target object is as determined by the name of the renamed entity.

6.1/3 {*AI05-0291-1*} A call on an entry or a protected subprogram either uses a name or prefix that determines a target object implicitly, as above, or is a call on (a non-prefixed view of) a primitive subprogram of a limited interface whose first parameter is a controlling parameter, in which case the target object is identified explicitly by the first parameter. This latter case is an *external call*.

7 A corresponding definition of target object applies to a requeue_statement (see 9.5.4), with a corresponding distinction between an *internal requeue* and an *external requeue*.

Legality Rules

7.1/3 {*AI95-00345-01*} {*AI05-0225-1*} {*AI05-0291-1*} If a name or prefix determines a target object, and the name denotes a protected entry or procedure, then the target object shall be a variable, unless the prefix is for an attribute_reference to the Count attribute (see 9.9).

7.a/3 **Reason:** {*AI05-0225-1*} The point is to prevent any calls to such a name whose target object is a constant view of a protected object, directly, or via an access value, renames, or generic formal subprogram. It is, however, legal to say P'Count in a protected function body, even though the protected object is a constant view there.

Ramification: {*AI05-0291-1*} This rule does not apply to calls that are not to a prefixed view. Specifically a "normal" call to a primitive operation of a limited interface is not covered by this rule. In that case, the normal parameter passing mode checks will prevent passing a constant protected object to an operation implemented by a protected entry or procedure as the mode is required to be **in out** or **out**. 7.b/3

{*AI12-0166-1*} An internal call on a protected function shall not occur within a precondition expression (see 6.1.1) of a protected operation nor within a default_expression of a parameter_specification of a protected operation. 7.2/5

Reason: {*AI125-0166-1*} These calls will be made before the start of the protected action, and thus would not be subject to the expected mutual exclusion. As such, they would be an automatic race condition (the state of the called object could change before the start of the protected action for the call on the protected entry or subprogram). 7.c/5

To be honest: {*AI125-0166-1*} 6.1.1 actually defines "specific precondition expression" and "class-wide precondition expression". This rule is intended to apply to both. 7.d/5

Dynamic Semantics

Within the body of a protected operation, the current instance (see 8.6) of the immediately enclosing protected unit is determined by the target object specified (implicitly or explicitly) in the call (or requeue) on the protected operation. 8

To be honest: The current instance is defined in the same way within the body of a subprogram declared immediately within a protected_body. 8.a

Any call on a protected procedure or entry of a target protected object is defined to be an update to the object, as is a requeue on such an entry. 9

Reason: Read/write access to the components of a protected object is granted while inside the body of a protected procedure or entry. Also, any protected entry call can change the value of the Count attribute, which represents an update. Any protected procedure call can result in servicing the entries, which again might change the value of a Count attribute. 9.a

Syntax

{*AI05-0030-2*} {*AI05-0215-1*} synchronization_kind ::= By_Entry | By_Protected_Procedure | Optional 10/3

Static Semantics

{*AI05-0215-1*} For the declaration of a primitive procedure of a synchronized tagged type the following language-defined representation aspect may be specified with an aspect_specification (see 13.1.1): 11/3

Synchronization 12/3

If specified, the aspect definition shall be a synchronization_kind.

Aspect Description for Synchronization: Defines whether a given primitive operation of a synchronized interface will be implemented by an entry or protected procedure. 12.a/5

{*AI05-0030-2*} {*AI05-0215-1*} Inherited subprograms inherit the Synchronization aspect, if any, from the corresponding subprogram of the parent or progenitor type. If an overriding operation does not have a directly specified Synchronization aspect then the Synchronization aspect of the inherited operation is inherited by the overriding operation. 13/3

Legality Rules

{*AI05-0030-2*} {*AI05-0215-1*} The synchronization_kind By_Protected_Procedure shall not be applied to a primitive procedure of a task interface. 14/3

{*AI05-0030-2*} {*AI05-0215-1*} A procedure for which the specified synchronization_kind is By_Entry shall be implemented by an entry. A procedure for which the specified synchronization_kind is By_Protected_Procedure shall be implemented by a protected procedure. A procedure for which the 15/3

specified synchronization_kind is Optional may be implemented by an entry or by a procedure (including a protected procedure).

16/3 {*AI05-0030-2*} {*AI05-0215-1*} If a primitive procedure overrides an inherited operation for which the Synchronization aspect has been specified to be By_Entry or By_Protected_Procedure, then any specification of the aspect Synchronization applied to the overriding operation shall have the same synchronization_kind.

17/3 {*AI05-0030-2*} In addition to the places where Legality Rules normally apply (see 12.3), these rules also apply in the private part of an instance of a generic unit.

Static Semantics

18/5 {*AI12-0064-2*} {*AI12-0374-2*} For a program unit, task entry, formal package, formal subprogram, formal object of an anonymous access-to-subprogram type, enumeration literal, and for a subtype (including a formal subtype), the following language-defined operational aspect is defined:

19/5 Nonblocking

This aspect specifies the blocking restriction for the entity; it shall be specified by a static Boolean expression. [The aspect_definition can be omitted from the specification of this aspect; in that case, the aspect for the entity is True.]

19.a/5 **Aspect Description for Nonblocking:** Specifies that an associated subprogram does not block.

19.b/5 **Proof:** 13.1.1 allows omitting the aspect expression for any aspect with type Boolean; we take advantage of that here.

20/5 {*AI12-0064-2*} The Nonblocking aspect may be specified for all entities for which it is defined, except for protected operations and task entries. In particular, Nonblocking may be specified for generic formal parameters.

20.a/5 **Ramification:** The Nonblocking aspect cannot be specified for predefined operators or enumeration literals but we don't need to mention that above. One would have to declare a subprogram in order to specify the aspect in those cases, but that defines a user-defined subprogram which is itself not a predefined operator or an enumeration literal.

21/5 {*AI12-0064-2*} {*AI12-0374-2*} {*AI12-0439-1*} When aspect Nonblocking is False for an entity, the entity can contain a potentially blocking operation; such an entity *allows blocking*. If the aspect is True for an entity, the entity is said to be *nonblocking*.

21.a/5 **Ramification:** Specifying Nonblocking as False imposes no restrictions. Specifying Nonblocking as True imposes additional compile-time checks to prevent blocking, but does not prevent deadlock. A pragma Detect_Blocking can be used to ensure that Program_Error is raised in a deadlock situation.

22/5 {*AI12-0064-2*} {*AI12-0374-2*} For a generic instantiation and entities declared within such an instance, the aspect is determined by the Nonblocking aspect for the corresponding entity of the generic unit, **and**ed with the Nonblocking aspects of the actual generic parameters *used* by the entity. If the aspect is directly specified for an instance, the specified expression shall have the same value as the Nonblocking aspect of the instance (after **and**ing with the aspects of the used actual parameters). In the absence of a Use_Formal aspect, all actual generic parameters are presumed to be *used* by an entity (see H.7.1).

22.a/5 **Reason:** We want to allow confirming aspects for instances, but nothing else. The Legality Rules of the generic body were checked assuming the Nonblocking aspect of the generic unit combined with the Nonblocking aspects of the formals where they are used, and if that is overridden on the instance, the instance body might make calls that allow blocking in subprograms that are declared nonblocking.

23/5 {*AI12-0064-2*} {*AI12-0374-2*} For a (protected or task) entry, the Nonblocking aspect is False.

23.a/5 **Reason:** An entry can be renamed as a procedure, so the value of the aspect has to be well-defined (as the attribute can be applied to a procedure). We do not want a nonblocking subprogram to be able to call an entry, no matter how it occurs, so the value ought to be False. Moreover, we do not want a subprogram that renames an entry to be able to override a nonblocking subprogram. We could have used individual rules for these cases, but there were already many of them, and this solution avoids the need for extra rules for entries.

{*AI12-0064-2*} {*AI12-0374-2*} For an enumeration literal, the Nonblocking aspect is True. 24/5

Reason: Enumeration literals can be renamed as functions, and passed to generic formal functions, so we need to define the value of the aspect to ensure the other rules are meaningful. 24.a/5

{*AI12-0064-2*} {*AI12-0374-2*} For a predefined operator of an elementary type, the Nonblocking aspect is True. For a predefined operator of a composite type, the Nonblocking aspect of the operator is the same as the Nonblocking aspect for the type. 25/5

Reason: Predefined operators of elementary types can never include any potentially blocking operations, so we want them to declare that. Record equality can be composed of operations including user-defined "=" operators, which might allow blocking. Array equality might use some record equality. So we have to have the possibility of allowing blocking for them. We don't just copy the Nonblocking aspect of the type in every case, as someone could declare an elementary type to allow blocking. 25.a/5

Ramification: It's not possible to specify the nonblocking expression of a predefined operator; if an operator is declared in order to do that, it is no longer predefined. 25.b/5

{*AI12-0064-2*} For a dereference of an access-to-subprogram type, the Nonblocking aspect of the designated subprogram is that of the access-to-subprogram type. 26/5

{*AI12-0374-2*} For the base subtype of a scalar (sub)type, the Nonblocking aspect is True. 27/5

Reason: The first subtype of a scalar type can allow blocking (which can be useful so a predicate can allow blocking), but the base subtype is always Nonblocking. We need this so the Nonblocking value is well-defined for any subtype that is built from the base subtype (T'Base). T'Base of any scalar type, including a generic formal type, is always nonblocking. 27.a/5

{*AI12-0064-2*} For an inherited primitive dispatching subprogram that is null or abstract, the subprogram is nonblocking if and only if a corresponding subprogram of at least one ancestor is nonblocking. For any other inherited subprogram, it is nonblocking if and only if the corresponding subprogram of the parent is nonblocking. 28/5

{*AI12-0064-2*} Unless directly specified, overridings of dispatching operations inherit this aspect. 29/5

{*AI12-0064-2*} {*AI12-0374-2*} Unless directly specified, for a formal subtype, formal package, or formal subprogram, the Nonblocking aspect is that of the actual subtype, package, or subprogram. 30/5

Reason: This means that Nonblocking legality checking for the actual parameters of the instance is only necessary when the aspect is explicitly specified for the formal type. 30.a/5

{*AI12-0064-2*} {*AI12-0374-2*} Unless directly specified, for a non-first subtype *S*, the Nonblocking aspect is that of the subtype identified in the subtype_indication defining *S*; unless directly specified for the first subtype of a derived type, the Nonblocking aspect is that of the ancestor subtype. 31/5

Discussion: The expressions that can be specified for a such a subtype are limited by a Legality Rule, see below. 31.a/5

{*AI12-0064-2*} Unless directly specified, for any other program unit, first subtype, or formal object, the Nonblocking aspect of the entity is determined by the Nonblocking aspect for the innermost program unit enclosing the entity. 32/5

{*AI12-0064-2*} {*AI12-0374-2*} If not specified for a library unit, the Nonblocking aspect is True if the library unit is declared pure, or False otherwise. 33/5

{*AI12-0064-2*} {*AI12-0247-1*} The following are defined to be *potentially blocking* operations: 34/5

Reason: The primary purpose of these rules is to define what operations are not allowed in a protected operation (blocking is not allowed). Some of these operations are not directly blocking. However, they are still treated as potentially blocking, because allowing them in a protected action might impose an undesirable implementation burden. 34.a/5

- a select_statement; 35/5
- an accept_statement; 36/5
- an entry_call_statement, or a call on a procedure that renames or is implemented by an entry; 37/5

38/5 • a delay_statement;

39/5 • an abort_statement;

40/5 • task creation or activation;

41/5 • during a protected action, an external call on a protected subprogram (or an external requeue) with the same target object as that of the protected action.

41.a/5 **Reason:** This is really a deadlocking call, rather than a blocking call, but we include it in this list for simplicity.

42/5 {*AI12-0064-2*} If a language-defined subprogram allows blocking, then a call on the subprogram is a potentially blocking operation.

42.a/5 **Ramification:** Calls on other subprograms that allow blocking are not themselves potentially blocking; the execution of the body could execute a potentially blocking operation.

42.b/5 A user-defined instance of a language-defined generic creates user-defined subprograms for the purpose of this rule. A dispatching call to a language-defined abstract subprogram always calls a user-defined concrete subprogram, so that too is not potentially blocking for the purposes of this rule.

Legality Rules

43/5 {*AI12-0064-2*} {*AI12-0267-1*} {*AI12-0374-2*} A portion of program text is called a *nonblocking region* if it is anywhere within a parallel construct, or if the innermost enclosing program unit is nonblocking. A nonblocking region shall not contain any of the following:

44/5 • a select_statement;

45/5 • an accept_statement;

46/5 • a delay_statement;

47/5 • an abort_statement;

48/5 • task creation or activation.

49/5 {*AI12-0374-2*} Furthermore, a parallel construct shall neither contain a call on a callable entity for which the Nonblocking aspect is False, nor shall it contain a call on a callable entity declared within a generic unit that uses a generic formal parameter with Nonblocking aspect False (see Use_Formal aspect in H.7.1).

50/5 {*AI12-0374-2*} Finally, a nonblocking region that is outside of a parallel construct shall not contain a call on a callable entity for which the Nonblocking aspect is False, unless the region is within a generic unit and the callable entity is associated with a generic formal parameter of the generic unit, or the call is within the aspect_definition of an assertion aspect for an entity that allows blocking.

50.a/5 **Reason:** A generic unit or entity declared within one is presumed to use its "used" generic formal parameters at least once each time it is invoked, and this passes through to the parallel construct check.

50.b/5 **Ramification:** Implicit calls for finalization, storage pools, and the like are covered by the above prohibition. The rules above say “a call”, not “an explicit call”. Such calls are considered statically bound when that is possible, that is, when the controlling object has a known specific type (even if the actual implementation uses dispatching).

50.c/5 **Discussion:** We don't need to worry specially about entry calls (even if the entry has been renamed as a procedure), as they will be detected by the prohibition against calls to entities with the Nonblocking aspect False.

50.d/5 Similarly, we don't need to specially worry about subprograms of limited interfaces that are implemented by entries, as any such subprogram necessarily has the value statically False for the Nonblocking aspect, and thus is already covered by the prohibition against calling such subprograms.

51/5 {*AI12-0064-2*} {*AI12-0374-2*} For the purposes of the above rules, an entry_body is considered nonblocking if the immediately enclosing protected unit is nonblocking.

51.a/5 **Reason:** An entry declaration always allows blocking (by rule); but we want to be able to compile-time check for most violations of the prohibition against potentially blocking operations in a protected action (see 9.5.1). We do that by using the nonblocking status of the protected unit as the controlling factor, and enforce that by not allowing the specification of the Nonblocking aspect for any protected operation. We can't do this checking unconditionally, as that

would be incompatible: existing Ada protected units might call subprograms that allow blocking. Thus a protected unit that allows blocking (which is the default) must allow calling any subprogram from an entry body.

{*AI12-0374-2*} For a subtype for which aspect Nonblocking is True, any predicate expression that applies to the subtype shall only contain constructs that are allowed immediately within a nonblocking program unit. 52/5

{*AI12-0064-2*} A subprogram shall be nonblocking if it overrides a nonblocking dispatching operation. An entry shall not implement a nonblocking procedure. If an inherited dispatching subprogram allows blocking, then the corresponding subprogram of each ancestor shall allow blocking. 53/5

Discussion: Rules elsewhere in the standard (4.6 and 3.10.2) ensure that access-to-subprogram conversion and the Access attribute enforce nonblocking. 53.a/5

Ramification: A nonblocking subprogram can override one that allows blocking, but the reverse is illegal. Thus one can declare a Finalize subprogram to be nonblocking, even though it overrides a routine that allows blocking. (This works because a nonblocking subprogram allows a strict subset of the operations allowed in allows blocking subprograms, so calling such a subprogram as if it allows blocking — as is necessary in a dispatching call — is harmless.) 53.b/5

{*AI12-0064-2*} {*AI12-0374-2*} {*AI12-0396-1*} {*AI12-0399-1*} It is illegal to directly specify aspect Nonblocking for the first subtype of the full view of a type that has a partial view. If the Nonblocking aspect of the full view is inherited, it shall have the same value as that of the partial view, or have the value True. 54/5

Reason: We need completions to agree on nonblocking with the original view. One reason this is necessary to prevent the predefined equality operator from being nonblocking in the partial view and allowing blocking in the full view. 54.a/5

{*AI12-0064-2*} {*AI12-0374-2*} Aspect Nonblocking shall be directly specified for the first subtype of a derived type only if it has the same value as the Nonblocking aspect of the ancestor subtype or if it is specified True. Aspect Nonblocking shall be directly specified for a nonfirst subtype *S* only if it has the same value as the Nonblocking aspect of the subtype identified in the subtype_indication defining *S* or if it is specified True. 55/5

Reason: Boolean-valued aspects have a similar rule to the first rule here (see 13.1.1), we want this one to work similarly. We need non-first subtypes to allow blocking only if the original first subtype allows blocking, as that allows the programmer to know that any operation on any subtype of a type are nonblocking if the first subtype is nonblocking. 55.a/5

{*AI12-0319-1*} For an access-to-object type that is nonblocking, the Allocate, Deallocate, and Storage_Size operations on its storage pool shall be nonblocking. 56/5

Ramification: Standard storage pools always have nonblocking operations by definition (see 13.11), so this rule only can fail for user-defined storage pools. 56.a/5

{*AI12-0319-1*} For a composite type that is nonblocking: 57/5

- All component subtypes shall be nonblocking; 58/5
- For a record type or extension, every call in the default_expression of a component (including discriminants) shall call an operation that is nonblocking; 59/5
- For a controlled type, the Initialize, Finalize, and Adjust (if any) subprograms shall be nonblocking. 60/5

Reason: These rules ensure that if a type is nonblocking, the default initialization, finalization, and assignment of the type are also nonblocking. This ensures that if a generic formal type is nonblocking, all of the basic operations of the actual type are nonblocking. 60.a/5

Default initialization, finalization, and assignment of elementary types are always nonblocking, so we don't need any rules for those. 60.b/5

{*AI12-0064-2*} {*AI12-0374-2*} The predefined equality operator for a composite type, unless it is for a record type or record extension and the operator is overridden by a primitive equality operator, is illegal if it is nonblocking and: 61/5

62/5 • for a record type or record extension, the parent primitive "=" allows blocking; or

63/5 • some component is of a type *T*, and:

64/5 • *T* is a record type or record extension that has a primitive "=" that allows blocking; or

65/5 • *T* is neither a record type nor a record extension, and *T* has a predefined "=" that allows blocking.

65.a/5 **Ramification:** This applies to both record and array "=".

65.b/5 This check occurs when the equality operator is declared, so this rule effectively makes the type illegal if the rule is violated.

65.c/5 **Reason:** We don't need to check this when the operator is overridden for a record type, as the body of the new definition of equality will enforce the rules, and there is no case where the predefined operator will re-emerge. We do have to check this for array types even if the operator is overridden, as the predefined operator will re-emerge in generics and record equality.

66/5 {*AI12-0064-2*} {*AI12-0374-2*} In a generic instantiation:

67/5 • the actual subprogram corresponding to a nonblocking formal subprogram shall be nonblocking [(an actual that is an entry is not permitted in this case)];

68/5 • the actual subtype corresponding to a nonblocking formal subtype shall be nonblocking;

68.a/5 **Ramification:** We require matching for formal scalar or access-to-object types, even though their predefined operators are always nonblocking (and they re-emerge in the generic unit) - because a "blocking" predicate might apply to the actual subtype, which will also be enforced on operations of the formal type.

69/5 • the actual object corresponding to a formal object of a nonblocking access-to-subprogram type shall be of a nonblocking access-to-subprogram type;

70/5 • the actual instance corresponding to a nonblocking formal package shall be nonblocking.

71/5 {*AI12-0064-2*} In addition to the places where Legality Rules normally apply (see 12.3), the above rules also apply in the private part of an instance of a generic unit.

71.a/5 **Ramification:** For a generic formal parameter to be nonblocking (thus, for these rules to apply), it has to explicitly specify aspect Nonblocking to be True. However, if not specified True, these rules do apply in the instance of the specification of the generic unit (the normal re-checking is needed). For instance, the body of an expression function might make a prohibited call.

72/3 NOTE {*AI05-0030-2*} {*AI05-0215-1*} The synchronization_kind By_Protected_Procedure implies that the operation will not block.

Wording Changes from Ada 95

72.a/2 {*AI95-00345-01*} Added a Legality Rule to make it crystal-clear that the protected object of an entry or procedure call must be a variable. This rule was implied by the Dynamic Semantics here, along with the Static Semantics of 3.3, but it is much better to explicitly say it. While many implementations have gotten this wrong, this is not an incompatibility — allowing updates of protected constants has always been wrong.

Extensions to Ada 2005

72.b/3 {*AI05-0030-2*} {*AI05-0215-1*} Added the Synchronization aspect to allow specifying that an interface procedure is really an entry or a protected procedure.

Wording Changes from Ada 2005

72.c/3 {*AI05-0225-1*} **Correction:** Clarified that the target object of any name denoted a protected procedure or entry can never be a constant (other than for the 'Count attribute). This closes holes involving calls to access-to-protected, renaming as a procedure, and generic formal subprograms.

Inconsistencies With Ada 2012

72.d/5 {*AI12-0064-2*} Calls on procedures that rename an entry or are implemented by an entry are now defined to be potentially blocking. This means that such a call now might raise Program_Error. However, it never made sense for some entry calls to be excluded from being potentially blocking, and we expect that most implementations already

treated all entry calls the same way. Thus do not expect this wording change to actually change the behavior of any implementation, and thus no program will change.

Incompatibilities With Ada 2012

{*AI12-0166-1*} **Correction:** Internal protected calls are now prohibited in preconditions and default expressions of protected operations. These were allowed in Ada 2012, but as they cause race conditions and as most existing Ada 95 compilers crash when given such a default parameter, we expect such code to be extremely rare. 72.e/5

Extensions to Ada 2012

{*AI12-0064-2*} {*AI12-0319-1*} {*AI12-0374-2*} Aspect Nonblocking is new; it allows compile-time checks to prevent using potentially blocking operations in contexts where that is not allowed. 72.f/5

9.5.1 Protected Subprograms and Protected Actions

A *protected subprogram* is a subprogram declared immediately within a protected_definition. Protected procedures provide exclusive read-write access to the data of a protected object; protected functions provide concurrent read-only access to the data. 1

Ramification: A subprogram declared immediately within a protected_body is not a protected subprogram; it is an intrinsic subprogram. See 6.3.1, “Conformance Rules”. 1.a

Static Semantics

[Within the body of a protected function (or a function declared immediately within a protected_body), the current instance of the enclosing protected unit is defined to be a constant (that is, its subcomponents may be read but not updated). Within the body of a protected procedure (or a procedure declared immediately within a protected_body), and within an entry_body, the current instance is defined to be a variable (updating is permitted).] 2

Proof: {*AI05-0120-1*} All constant views are defined in 3.3, “Objects and Named Numbers”, anything not named there is a variable view. 2.a.1/3

Ramification: The current instance is like an implicit parameter, of mode **in** for a protected function, and of mode **in out** for a protected procedure (or protected entry). 2.a

{*AI12-0129-1*} For a type declared by a protected_type_declaration or for the anonymous type of an object declared by a single_protected_declaration, the following language-defined type-related representation aspect may be specified: 2.1/4

Exclusive_Functions 2.2/4

The type of aspect Exclusive_Functions is Boolean. If not specified (including by inheritance), the aspect is False.

A value of True for this aspect indicates that protected functions behave in the same way as protected procedures with respect to mutual exclusion and queue servicing (see below). 2.3/4

Aspect Description for Exclusive_Functions: Specifies mutual exclusion behavior of protected functions in a protected type. 2.a.1/4

{*AI12-0129-1*} A protected procedure or entry is an *exclusive* protected operation. A protected function of a protected type *P* is an exclusive protected operation if the Exclusive_Functions aspect of *P* is True. 2.4/4

Dynamic Semantics

For the execution of a call on a protected subprogram, the evaluation of the name or prefix and of the parameter associations, and any assigning back of **in out** or **out** parameters, proceeds as for a normal subprogram call (see 6.4). If the call is an internal call (see 9.5), the body of the subprogram is executed as for a normal subprogram call. If the call is an external call, then the body of the subprogram is executed as part of a new *protected action* on the target protected object; the protected action completes 3

after the body of the subprogram is executed. [A protected action can also be started by an entry call (see 9.5.3).]

4/4 {*AI12-0129-1*} A new protected action is not started on a protected object while another protected action on the same protected object is underway, unless both actions are the result of a call on a nonexclusive protected function. This rule is expressible in terms of the execution resource associated with the protected object:

5/4 - {*AI12-0129-1*} *Starting* a protected action on a protected object corresponds to *acquiring* the execution resource associated with the protected object, either for exclusive read-write access if the protected action is for a call on an exclusive protected operation, or for concurrent read-only access otherwise;

6 - *Completing* the protected action corresponds to *releasing* the associated execution resource.

7/4 {*AI12-0129-1*} [After performing an exclusive protected operation on a protected object, but prior to completing the associated protected action, the entry queues (if any) of the protected object are serviced (see 9.5.3).]

7.1/5 {*AI12-0119-1*} If a parallel construct occurs within a protected action, no new logical threads of control are created. Instead, each element of the parallel construct that would have become a separate logical thread of control executes on the logical thread of control that is performing the protected action. If there are multiple such elements initiated at the same point, they execute in an arbitrary order.

7.a/5 **Reason:** It would be feasible to allow multiple logical threads of control within a protected action, but it would significantly complicate the definition of "sequential" and "concurrent" actions, since we generally presume that everything occuring within protected actions of a given protected object is sequential. We could simply disallow any use of parallel constructs, but that seems unnecessary, particularly as a parallel construct might be buried within a subprogram that is declared Nonblocking.

Bounded (Run-Time) Errors

8/5 {*AI12-0064-2*} During a protected action, it is a bounded error to invoke an operation that is potentially blocking (see 9.5).

Paragraphs 9 through 16 were moved to 9.5.

17/5 {*AI12-0439-1*} If the bounded error is detected, Program_Error is raised. If not detected, the bounded error can result in deadlock or a (nested) protected action on the same target object.

17.a/2 **Discussion:** {*AI95-00305-01*} By "nested protected action", we mean that an additional protected action can be started by another task on the same protected object. This means that mutual exclusion may be broken in this bounded error case. A way to ensure that this does not happen is to use pragma Detect_Blocking (see H.5).

18/5 {*AI12-0064-2*} {*AI12-0247-1*} During a protected action, a call on a subprogram whose body contains a potentially blocking operation is a bounded error. If the bounded error is detected, Program_Error is raised; otherwise, the call proceeds normally.

18.b/5 **Reason:** {*AI12-0247-1*} This allows an implementation to check and raise Program_Error as soon as a subprogram is called, rather than waiting to find out whether it actually reaches the potentially blocking operation. If the call proceeds normally, reaching the potentially blocking operation is a separate bounded error, covered by the previous rules.

19/5 NOTE 1 {*AI12-0276-1*} {*AI12-0440-1*} If two tasks both try to start a protected action on a protected object, and at most one is calling a protected nonexclusive function, then only one of the tasks can proceed. Although the other task cannot proceed, it is not considered blocked, and it can be consuming processing resources while it awaits its turn. Unless there is an admission policy (see D.4.1) in effect, there is no language-defined ordering or queuing presumed for tasks competing to start a protected action — on a multiprocessor such tasks can use busy-waiting; for further monoprocessor and multiprocessor considerations, see D.3, "Priority Ceiling Locking".

19.a **Discussion:** The intended implementation on a multi-processor is in terms of "spin locks" — the waiting task will spin.

20/5 NOTE 2 {*AI12-0440-1*} The body of a protected unit can contain declarations and bodies for local subprograms. These are not visible outside the protected unit.

NOTE 3 The body of a protected function can contain internal calls on other protected functions, but not protected procedures, because the current instance is a constant. On the other hand, the body of a protected procedure can contain internal calls on both protected functions and procedures. 21

NOTE 4 From within a protected action, an internal call on a protected subprogram, or an external call on a protected subprogram with a different target object is not considered a potentially blocking operation. 22

Reason: This is because a task is not considered blocked while attempting to acquire the execution resource associated with a protected object. The acquisition of such a resource is rather considered part of the normal competition for execution resources between the various tasks that are ready. External calls that turn out to be on the same target object are considered potentially blocking, since they can deadlock the task indefinitely. 22.a

NOTE 5 {*AI95-00305-01*} {*AI12-0064-2*} {*AI12-0440-1*} The aspect Nonblocking can be specified True on the definition of a protected unit in order to reject most attempts to use potentially blocking operations within the protected unit (see 9.5). The pragma Detect_Blocking can be used to ensure that any remaining executions of potentially blocking operations during a protected action raise Program_Error. See H.5. 23/5

Discussion: {*AI12-0064-2*} The deadlock case cannot be detected at compile-time, so pragma Detect_Blocking is needed to give it consistent behavior. 23.a/5

Examples

Examples of protected subprogram calls (see 9.4): 24

```
Shared_Array.Set_Component(N, E);                                    25
E := Shared_Array.Component(M);
Control.Release;
```

Wording Changes from Ada 95

{*AI95-00305-01*} Added a note pointing out the existence of pragma Detect_Blocking. This pragma can be used to ensure portable (somewhat pessimistic) behavior of protected actions by converting the Bounded Error into a required check. 25.a/2

Extensions to Ada 2012

{*AI12-0129-1*} **Corrigendum:** Aspect Exclusive_Functions is new. The term “exclusive protected operations” is new. 25.b/4

Wording Changes from Ada 2012

{*AI12-0064-2*} Moved the definition of potentially blocking operations to 9.5, so it could be integrated into the definition of the Nonblocking aspect. 25.c/5

{*AI12-0247-1*} **Correction:** Added a separate bounded error for a subprogram containing a blocking operation, to keep compatibility with Ada 95 rules without requiring a correct implementation of pragma Detect_Blocking to do full program analysis. 25.d/5

9.5.2 Entries and Accept Statements

Entry_declarations, with the corresponding entry_bodies or accept_statements, are used to define potentially queued operations on tasks and protected objects. 1

Syntax

{*AI95-00397-01*} {*AI05-0183-1*} entry_declaration ::= 2/3
  [overriding_indicator]
  entry defining_identifier [(discrete_subtype_definition)] parameter_profile
    [aspect_specification];

accept_statement ::= 3
  accept *entry*_direct_name [(entry_index)] parameter_profile [**do**
    handled_sequence_of_statements
  end [*entry*_identifier]];

3.a **Reason:** We cannot use defining_identifier for accept_statements. Although an accept_statement is sort of like a body, it can appear nested within a block_statement, and therefore be hidden from its own entry by an outer homograph.

4 entry_index ::= expression

5/5 {*AI12-0169-1*} entry_body ::=
entry defining_identifier entry_body_formal_part
[aspect_specification]
entry_barrier **is**
declarative_part
begin
handled_sequence_of_statements
end [*entry*_identifier];

5.a/2 **Discussion:** {*AI95-00397-01*} We don't allow an overriding_indicator on an entry_body because entries always implement procedures at the point of the type declaration; there is no late implementation. And we don't want to have to think about overriding_indicators on accept_statements.

6 entry_body_formal_part ::= [(entry_index_specification)] parameter_profile

7 entry_barrier ::= **when** condition

8/5 {*AI12-0398-1*} entry_index_specification ::=
for defining_identifier **in** discrete_subtype_definition [aspect_specification]

9 If an *entry*_identifier appears at the end of an accept_statement, it shall repeat the *entry*_direct_-name. If an *entry*_identifier appears at the end of an entry_body, it shall repeat the defining_-identifier.

10 [An entry_declaration is allowed only in a protected or task declaration.]

10.a **Proof:** This follows from the BNF.

10.1/2 {*AI95-00397-01*} An overriding_indicator is not allowed in an entry_declaration that includes a discrete_subtype_definition.

10.a.1/2 **Reason:** An entry family can never implement something, so allowing an indicator is felt by the majority of the ARG to be redundant.

Name Resolution Rules

11 In an accept_statement, the expected profile for the *entry*_direct_name is that of the entry_declaration; the expected type for an entry_index is that of the subtype defined by the discrete_subtype_definition of the corresponding entry_declaration.

12 Within the handled_sequence_of_statements of an accept_statement, if a selected_component has a prefix that denotes the corresponding entry_declaration, then the entity denoted by the prefix is the accept_statement, and the selected_component is interpreted as an expanded name (see 4.1.3)[; the selector_name of the selected_component has to be the identifier for some formal parameter of the accept_statement].

12.a **Proof:** The only declarations that occur immediately within the declarative region of an accept_statement are those for its formal parameters.

Legality Rules

13 An entry_declaration in a task declaration shall not contain a specification for an access parameter (see 3.10).

13.a **Reason:** Access parameters for task entries would require a complex implementation. For example:

```
task T is                                                                    13.b
   entry E(Z : access Integer); -- Illegal!
end T;
task body T is                                                               13.c
begin
   declare
      type A is access all Integer;
      X : A;
      Int : aliased Integer;
      task Inner;
      task body Inner is
      begin
         T.E(Int'Access);
      end Inner;
   begin
      accept E(Z : access Integer) do
         X := A(Z); -- Accessibility_Check
      end E;
   end;
end T;
```

Implementing the Accessibility_Check inside the accept_statement for E is difficult, since one does not know whether the entry caller is calling from inside the immediately enclosing declare block or from outside it. This means that the lexical nesting level associated with the designated object is not sufficient to determine whether the Accessibility_Check should pass or fail. 13.d

Note that such problems do not arise with protected entries, because entry_bodies are always nested immediately within the protected_body; they cannot be further nested as can accept_statements, nor can they be called from within the protected_body (since no entry calls are permitted inside a protected_body). 13.e

{*AI95-00397-01*} If an entry_declaration has an overriding_indicator, then at the point of the declaration: 13.1/2

- if the overriding_indicator is **overriding**, then the entry shall implement an inherited subprogram; 13.2/2
- if the overriding_indicator is **not overriding**, then the entry shall not implement any inherited subprogram. 13.3/2

In addition to the places where Legality Rules normally apply (see 12.3), these rules also apply in the private part of an instance of a generic unit. 13.4/2

Discussion: These rules are subtly different than those for subprograms (see 8.3.1) because there cannot be "late" inheritance of primitives from interfaces. Hidden (that is, private) interfaces are prohibited explicitly (see 7.3), as are hidden primitive operations (as private operations of public abstract types are prohibited — see 3.9.3). 13.f/2

For an accept_statement, the innermost enclosing body shall be a task_body, and the *entry*_direct_-name shall denote an entry_declaration in the corresponding task declaration; the profile of the accept_-statement shall conform fully to that of the corresponding entry_declaration. An accept_statement shall have a parenthesized entry_index if and only if the corresponding entry_declaration has a discrete_subtype_definition. 14

An accept_statement shall not be within another accept_statement that corresponds to the same entry_declaration, nor within an asynchronous_select inner to the enclosing task_body. 15

Reason: Accept_statements are required to be immediately within the enclosing task_body (as opposed to being in a nested subprogram) to ensure that a nested task does not attempt to accept the entry of its enclosing task. We considered relaxing this restriction, either by making the check a run-time check, or by allowing a nested task to accept an entry of its enclosing task. However, neither change seemed to provide sufficient benefit to justify the additional implementation burden. 15.a

Nested accept_statements for the same entry (or entry family) are prohibited to ensure that there is no ambiguity in the resolution of an expanded name for a formal parameter of the entry. This could be relaxed by allowing the inner one to hide the outer one from all visibility, but again the small added benefit didn't seem to justify making the change for Ada 95. 15.b

Accept_statements are not permitted within asynchronous_select statements to simplify the semantics and implementation: an accept_statement in an abortable_part could result in Tasking_Error being propagated from an entry call even though the target task was still callable; implementations that use multiple tasks implicitly to implement 15.c

an asynchronous_select might have trouble supporting "up-level" accepts. Furthermore, if accept_statements were permitted in the abortable_part, a task could call its own entry and then accept it in the abortable_part, leading to rather unusual and possibly difficult-to-specify semantics.

16 An entry_declaration of a protected unit requires a completion[, which shall be an entry_body,] and every entry_body shall be the completion of an entry_declaration of a protected unit. The profile of the entry_body shall conform fully to that of the corresponding declaration.

16.a **Ramification:** An entry_declaration, unlike a subprogram_declaration, cannot be completed with a renaming_-declaration.

16.b/3 **To be honest:** {*AI05-0229-1*} If the implementation supports it, the entry body can be imported (using aspect Import, see B.1), in which case no explicit entry_body is allowed.

16.c **Discussion:** The above applies only to protected entries, which are the only ones completed with entry_bodies. Task entries have corresponding accept_statements instead of having entry_bodies, and we do not consider an accept_statement to be a "completion," because a task entry_declaration is allowed to have zero, one, or more than one corresponding accept_statements.

17 An entry_body_formal_part shall have an entry_index_specification if and only if the corresponding entry_declaration has a discrete_subtype_definition. In this case, the discrete_subtype_definitions of the entry_declaration and the entry_index_specification shall fully conform to one another (see 6.3.1).

18 A name that denotes a formal parameter of an entry_body is not allowed within the entry_barrier of the entry_body.

Static Semantics

19 The parameter modes defined for parameters in the parameter_profile of an entry_declaration are the same as for a subprogram_declaration and have the same meaning (see 6.2).

19.a **Discussion:** Note that access parameters are not allowed for task entries (see above).

20 An entry_declaration with a discrete_subtype_definition (see 3.6) declares a *family* of distinct entries having the same profile, with one such entry for each value of the *entry index subtype* defined by the discrete_subtype_definition. [A name for an entry of a family takes the form of an indexed_component, where the prefix denotes the entry_declaration for the family, and the index value identifies the entry within the family.] The term *single entry* is used to refer to any entry other than an entry of an entry family.

21 In the entry_body for an entry family, the entry_index_specification declares a named constant whose subtype is the entry index subtype defined by the corresponding entry_declaration; the value of the *named entry index* identifies which entry of the family was called.

21.a **Ramification:** The discrete_subtype_definition of the entry_index_specification is not elaborated; the subtype of the named constant declared is defined by the discrete_subtype_definition of the corresponding entry_declaration, which is elaborated, either when the type is declared, or when the object is created, if its constraint is per-object.

Dynamic Semantics

22/1 {*8652/0002*} {*AI95-00171-01*} The elaboration of an entry_declaration for an entry family consists of the elaboration of the discrete_subtype_definition, as described in 3.8. The elaboration of an entry_-declaration for a single entry has no effect.

22.a/5 **Discussion:** {*AI05-0299-1*} {*AI12-0449-1*} The elaboration of the declaration of a protected subprogram has no effect, as specified in 6.1. The default initialization of an object of a task or protected type is covered in 3.3.1.

23 [The actions to be performed when an entry is called are specified by the corresponding accept_-statements (if any) for an entry of a task unit, and by the corresponding entry_body for an entry of a protected unit.]

23.1/5 {*AI12-0193-1*} The interaction between a task that calls an entry and an accepting task is called a *rendezvous*.

{*AI12-0193-1*} For the execution of an accept_statement, the entry_index, if any, is first evaluated and converted to the entry index subtype; this index value identifies which entry of the family is to be accepted. Further execution of the accept_statement is then blocked until a caller of the corresponding entry is selected (see 9.5.3), whereupon the handled_sequence_of_statements, if any, of the accept_statement is executed, with the formal parameters associated with the corresponding actual parameters of the selected entry call. Execution of the rendezvous consists of the execution of the handled_sequence_of_statements, performance of any postcondition or type invariant checks associated with the entry, and any initialization or finalization associated with these checks, as described in 6.1.1 and 7.3.2. After execution of the rendezvous, the accept_statement completes and is left. [The two tasks then proceed independently.] When an exception is propagated from the handled_sequence_of_statements of an accept_statement, the same exception is also raised by the execution of the corresponding entry_call_statement. 24/5

Ramification: This is in addition to propagating it to the construct containing the accept_statement. In other words, for a rendezvous, the raising splits in two, and continues concurrently in both tasks. 24.a

The caller gets a new occurrence; this isn't considered propagation. 24.b

Note that we say "propagated from the handled_sequence_of_statements of an accept_statement", not "propagated from an accept_statement". The latter would be wrong — we don't want exceptions propagated by the entry_index to be sent to the caller (there is none yet!). 24.c

{*AI12-0193-1*} Execution of the rendezvous does not include any checks associated with parameter copy back or any post-call subtype predicate check for a parameter which is passed by reference. These checks are performed by the caller after the execution of the rendezvous. 24.d/5

This paragraph was deleted.{*AI12-0193-1*} 25/5

[An entry_body is executed when the condition of the entry_barrier evaluates to True and a caller of the corresponding single entry, or entry of the corresponding entry family, has been selected (see 9.5.3).] For the execution of the entry_body, the declarative_part of the entry_body is elaborated, and the handled_-sequence_of_statements of the body is executed, as for the execution of a subprogram_body. The value of the named entry index, if any, is determined by the value of the entry index specified in the *entry*_name of the selected entry call (or intermediate requeue_statement — see 9.5.4). 26

To be honest: If the entry had been renamed as a subprogram, and the call was a procedure_call_statement using the name declared by the renaming, the entry index (if any) comes from the entry name specified in the subprogram_renaming_declaration. 26.a

NOTE 1 A task entry has corresponding accept_statements (zero or more), whereas a protected entry has a corresponding entry_body (exactly one). 27

NOTE 2 A consequence of the rule regarding the allowed placements of accept_statements is that a task can execute accept_statements only for its own entries. 28

NOTE 3 {*AI95-00318-02*} {*AI12-0440-1*} A return statement (see 6.5) or a requeue_statement (see 9.5.4) can be used to complete the execution of an accept_statement or an entry_body. 29/5

Ramification: An accept_statement need not have a handled_sequence_of_statements even if the corresponding entry has parameters. Equally, it can have a handled_sequence_of_statements even if the corresponding entry has no parameters. 29.a

Ramification: A single entry overloads a subprogram, an enumeration literal, or another single entry if they have the same defining_identifier. Overloading is not allowed for entry family names. A single entry or an entry of an entry family can be renamed as a procedure as explained in 8.5.4. 29.b

NOTE 4 {*AI12-0440-1*} The condition in the entry_barrier can reference anything visible except the formal parameters of the entry. This includes the entry index (if any), the components (including discriminants) of the protected object, the Count attribute of an entry of that protected object, and data global to the protected unit. 30/5

The restriction against referencing the formal parameters within an entry_barrier ensures that all calls of the same entry see the same barrier value. If it is necessary to look at the parameters of an entry call before deciding whether to handle it, the entry_barrier can be "**when** True" and the caller can be requeued (on some private entry) when its parameters indicate that it cannot be handled immediately. 31

Examples

32 *Examples of entry declarations:*

33
```
entry Read(V : out Item);
entry Seize;
entry Request(Level)(D : Item);   -- a family of entries
```

34 *Examples of accept statements:*

35
```
accept Shut_Down;
```

36
```
accept Read(V : out Item) do
   V := Local_Item;
end Read;
```

37
```
accept Request(Low)(D : Item) do
   ...
end Request;
```

Extensions to Ada 83

37.a The syntax rule for entry_body is new.

37.b Accept_statements can now have exception_handlers.

Wording Changes from Ada 95

37.c/2 {*8652/0002*} {*AI95-00171-01*} **Corrigendum:** Clarified the elaboration of per-object constraints.

37.d/2 {*AI95-00397-01*} Overriding_indicators can be used on entries; this is only useful when a task or protected type inherits from an interface.

Extensions to Ada 2005

37.e/3 {*AI05-0183-1*} An optional aspect_specification can be used in an entry_declaration. This is described in 13.1.1.

Extensions to Ada 2012

37.f/5 {*AI12-0169-1*} **Correction:** An optional aspect_specification can be used in an entry_body. All other kinds of bodies allow (only) implementation-defined aspects, we need to be consistent.

37.g/5 {*AI12-0398-1*} Named entry indexes now can have an aspect_specification, allowing the specification of (implementation-defined) aspects for a named entry index.

Wording Changes from Ada 2012

37.h/5 {*AI12-0193-1*} **Correction:** Clarified that postcondition and invariant checks are clearly part of the rendezvous for an entry call. 6.1.1 already said this, so the intent was clear and this is not an inconsistency.

9.5.3 Entry Calls

1 [An entry_call_statement (an *entry call*) can appear in various contexts.] A *simple* entry call is a stand-alone statement that represents an unconditional call on an entry of a target task or a protected object. [Entry calls can also appear as part of select_statements (see 9.7).]

Syntax

2 entry_call_statement ::= *entry_*name [actual_parameter_part];

Name Resolution Rules

3 The *entry_*name given in an entry_call_statement shall resolve to denote an entry. The rules for parameter associations are the same as for subprogram calls (see 6.4 and 6.4.1).

Static Semantics

[The *entry_*name of an entry_call_statement specifies (explicitly or implicitly) the target object of the call, the entry or entry family, and the entry index, if any (see 9.5).] 4

Dynamic Semantics

Under certain circumstances (detailed below), an entry of a task or protected object is checked to see whether it is *open* or *closed*: 5

- {*AI05-0264-1*} An entry of a task is open if the task is blocked on an accept_statement that corresponds to the entry (see 9.5.2), or on a selective_accept (see 9.7.1) with an open accept_alternative that corresponds to the entry; otherwise, it is closed. 6/3
- {*AI05-0264-1*} An entry of a protected object is open if the condition of the entry_barrier of the corresponding entry_body evaluates to True; otherwise, it is closed. If the evaluation of the condition propagates an exception, the exception Program_Error is propagated to all current callers of all entries of the protected object. 7/3

  Reason: An exception during barrier evaluation is considered essentially a fatal error. All current entry callers are notified with a Program_Error. In a fault-tolerant system, a protected object might provide a Reset protected procedure, or equivalent, to support attempts to restore such a "broken" protected object to a reasonable state. 7.a

  Discussion: Note that the definition of when a task entry is open is based on the state of the (accepting) task, whereas the "openness" of a protected entry is defined only when it is explicitly checked, since the barrier expression needs to be evaluated. Implementation permissions are given (below) to allow implementations to evaluate the barrier expression more or less often than it is checked, but the basic semantic model presumes it is evaluated at the times when it is checked. 7.b

For the execution of an entry_call_statement, evaluation of the name and of the parameter associations is as for a subprogram call (see 6.4). The entry call is then *issued*: For a call on an entry of a protected object, a new protected action is started on the object (see 9.5.1). The named entry is checked to see if it is open; if open, the entry call is said to be *selected immediately*, and the execution of the call proceeds as follows: 8

- For a call on an open entry of a task, the accepting task becomes ready and continues the execution of the corresponding accept_statement (see 9.5.2). 9
- For a call on an open entry of a protected object, the corresponding entry_body is executed (see 9.5.2) as part of the protected action. 10

If the accept_statement or entry_body completes other than by a requeue (see 9.5.4), return is made to the caller (after servicing the entry queues — see below); any necessary assigning back of formal to actual parameters occurs, as for a subprogram call (see 6.4.1); such assignments take place outside of any protected action. 11

Ramification: The return to the caller will generally not occur until the protected action completes, unless some other thread of control is given the job of completing the protected action and releasing the associated execution resource. 11.a

If the named entry is closed, the entry call is added to an *entry queue* (as part of the protected action, for a call on a protected entry), and the call remains queued until it is selected or cancelled; there is a separate (logical) entry queue for each entry of a given task or protected object [(including each entry of an entry family)]. 12

When a queued call is *selected*, it is removed from its entry queue. Selecting a queued call from a particular entry queue is called *servicing* the entry queue. An entry with queued calls can be serviced under the following circumstances: 13

- When the associated task reaches a corresponding accept_statement, or a selective_accept with a corresponding open accept_alternative; 14

15/4 • {*AI12-0129-1*} If after performing, as part of a protected action on the associated protected object, an exclusive protected operation on the object, the entry is checked and found to be open.

16 If there is at least one call on a queue corresponding to an open entry, then one such call is selected according to the *entry queuing policy* in effect (see below), and the corresponding accept_statement or entry_body is executed as above for an entry call that is selected immediately.

17 The entry queuing policy controls selection among queued calls both for task and protected entry queues. The default entry queuing policy is to select calls on a given entry queue in order of arrival. If calls from two or more queues are simultaneously eligible for selection, the default entry queuing policy does not specify which queue is serviced first. Other entry queuing policies can be specified by pragmas (see D.4).

18 For a protected object, the above servicing of entry queues continues until there are no open entries with queued calls, at which point the protected action completes.

18.a **Discussion:** While servicing the entry queues of a protected object, no new calls can be added to any entry queue of the object, except due to an internal requeue (see 9.5.4). This is because the first step of a call on a protected entry is to start a new protected action, which implies acquiring (for exclusive read-write access) the execution resource associated with the protected object, which cannot be done while another protected action is already in progress.

19 For an entry call that is added to a queue, and that is not the triggering_statement of an asynchronous_-select (see 9.7.4), the calling task is blocked until the call is cancelled, or the call is selected and a corresponding accept_statement or entry_body completes without requeuing. In addition, the calling task is blocked during a rendezvous.

19.a **Ramification:** For a call on a protected entry, the caller is not blocked if the call is selected immediately, unless a requeue causes the call to be queued.

20 An attempt can be made to cancel an entry call upon an abort (see 9.8) and as part of certain forms of select_statement (see 9.7.2, 9.7.3, and 9.7.4). The cancellation does not take place until a point (if any) when the call is on some entry queue, and not protected from cancellation as part of a requeue (see 9.5.4); at such a point, the call is removed from the entry queue and the call completes due to the cancellation. The cancellation of a call on an entry of a protected object is a protected action[, and as such cannot take place while any other protected action is occurring on the protected object. Like any protected action, it includes servicing of the entry queues (in case some entry barrier depends on a Count attribute).]

20.a/2 **Implementation Note:** {*AI95-00114-01*} In the case of an attempted cancellation due to abort, this removal might have to be performed by the calling task itself if the ceiling priority of the protected object is lower than the priority of the task initiating the abort.

21 A call on an entry of a task that has already completed its execution raises the exception Tasking_Error at the point of the call; similarly, this exception is raised at the point of the call if the called task completes its execution or becomes abnormal before accepting the call or completing the rendezvous (see 9.8). This applies equally to a simple entry call and to an entry call as part of a select_statement.

Implementation Permissions

22/5 {*AI12-0444-1*} An implementation may perform the sequence of steps of a protected action using any thread of control; it can be a thread other than that of the task that started the protected action. If an entry_body completes without requeuing, then the corresponding calling task may be made ready without waiting for the entire protected action to complete.

22.a **Reason:** These permissions are intended to allow flexibility for implementations on multiprocessors. On a monoprocessor, which thread of control executes the protected action is essentially invisible, since the thread is not abortable in any case, and the "current_task" function is not guaranteed to work during a protected action (see C.7.1).

23/5 {*AI12-0129-1*} {*AI12-0444-1*} When the entry of a protected object is checked to see whether it is open, the implementation can bypass reevaluating the condition of the corresponding entry_barrier if no variable or attribute referenced by the condition (directly or indirectly) has been altered by the execution

(or cancellation) of a call to an exclusive protected operation of the object since the condition was last evaluated.

Ramification: Changes to variables referenced by an entry barrier that result from actions outside of a call to an exclusive protected operation of the protected object need not be "noticed". For example, if a global variable is referenced by an entry barrier, it should not be altered (except as part of a protected action on the object) any time after the barrier is first evaluated. In other words, globals can be used to "parameterize" a protected object, but they cannot reliably be used to control it after the first use of the protected object. 23.a/4

Implementation Note: Note that even if a global variable is volatile, the implementation need only reevaluate a barrier if the global is updated during a protected action on the protected object. This ensures that an entry-open bit-vector implementation approach is possible, where the bit-vector is computed at the end of a protected action, rather than upon each entry call. 23.b

An implementation may evaluate the conditions of all entry_barriers of a given protected object any time any entry of the object is checked to see if it is open. 24

Ramification: In other words, any side effects of evaluating an entry barrier should be innocuous, since an entry barrier might be evaluated more or less often than is implied by the "official" dynamic semantics. 24.a

Implementation Note: It is anticipated that when the number of entries is known to be small, all barriers will be evaluated any time one of them needs to be, to produce an "entry-open bit-vector". The appropriate bit will be tested when the entry is called, and only if the bit is false will a check be made to see whether the bit-vector might need to be recomputed. This should allow an implementation to maximize the performance of a call on an open entry, which seems like the most important case. 24.b

In addition to the entry-open bit-vector, an "is-valid" bit is needed per object, which indicates whether the current bit-vector setting is valid. A "depends-on-Count-attribute" bit is needed per type. The "is-valid" bit is set to false (as are all the bits of the bit-vector) when the protected object is first created, as well as any time an exception is propagated from computing the bit-vector. Is-valid would also be set false any time the Count is changed and "depends-on-Count-attribute" is true for the type, or a protected procedure or entry returns indicating it might have updated a variable referenced in some barrier. 24.c

A single procedure can be compiled to evaluate all of the barriers, set the entry-open bit-vector accordingly, and set the is-valid bit to true. It could have a "when others" handler to set them all false, and call a routine to propagate Program_Error to all queued callers. 24.d

For protected types where the number of entries is not known to be small, it makes more sense to evaluate a barrier only when the corresponding entry is checked to see if it is open. It isn't worth saving the state of the entry between checks, because of the space that would be required. Furthermore, the entry queues probably want to take up space only when there is actually a caller on them, so rather than an array of all entry queues, a linked list of nonempty entry queues make the most sense in this case, with the first caller on each entry queue acting as the queue header. 24.e

{*AI12-0444-1*} When an attempt is made to cancel an entry call, the implementation can use a thread of control other than that of the task (or interrupt) that initiated the cancellation; in particular, it may use the thread of control of the caller itself to attempt the cancellation, even if this can allow the entry call to be selected in the interim. 25/5

Reason: Because cancellation of a protected entry call is a protected action (which helps make the Count attribute of a protected entry meaningful), it might not be practical to attempt the cancellation from the thread of control that initiated the cancellation. For example, if the cancellation is due to the expiration of a delay, it is unlikely that the handler of the timer interrupt could perform the necessary protected action itself (due to being on the interrupt level). Similarly, if the cancellation is due to an abort, it is possible that the task initiating the abort has a priority higher than the ceiling priority of the protected object (for implementations that support ceiling priorities). Similar considerations could apply in a multiprocessor situation. 25.a

NOTE 1 If an exception is raised during the execution of an entry_body, it is propagated to the corresponding caller (see 11.4). 26

NOTE 2 For a call on a protected entry, the entry is checked to see if it is open prior to queuing the call, and again thereafter if its Count attribute (see 9.9) is referenced in some entry barrier. 27

Ramification: Given this, extra care is required if a reference to the Count attribute of an entry appears in the entry's own barrier. 27.a

Reason: An entry is checked to see if it is open prior to queuing to maximize the performance of a call on an open entry. 27.b

NOTE 3 In addition to simple entry calls, the language permits timed, conditional, and asynchronous entry calls (see 9.7.2, 9.7.3, and see 9.7.4). 28

28.a **Ramification:** A task can call its own entries, but the task will deadlock if the call is a simple entry call.

29/5 NOTE 4 {*AI12-0442-1*} The condition of an entry_barrier is allowed to be evaluated by an implementation more often than strictly necessary, even if the evaluation can have side effects. On the other hand, an implementation can avoid reevaluating the condition if nothing it references was updated by an intervening protected action on the protected object, even if the condition references some global variable that is updated by an action performed from outside of a protected action.

Examples

30 *Examples of entry calls:*

31
```
Agent.Shut_Down;                        -- see 9.1
Parser.Next_Lexeme(E);                  -- see 9.1
Pool(5).Read(Next_Char);                -- see 9.1
Controller.Request(Low)(Some_Item);     -- see 9.1
Flags(3).Seize;                         -- see 9.4
```

Wording Changes from Ada 2012

31.a/4 {*AI12-0129-1*} **Corrigendum:** Revised wording to talk about "exclusive protected operations" (see 9.5.1).

9.5.4 Requeue Statements

1 [A requeue_statement can be used to complete an accept_statement or entry_body, while redirecting the corresponding entry call to a new (or the same) entry queue. Such a *requeue* can be performed with or without allowing an intermediate cancellation of the call, due to an abort or the expiration of a delay.]

Syntax

2/3 {*AI05-0030-2*} requeue_statement ::= **requeue** *procedure_or_entry_*name [**with abort**];

Name Resolution Rules

3/3 {*AI05-0030-2*} {*AI05-0215-1*} The *procedure_or_entry_*name of a requeue_statement shall resolve to denote a procedure or an entry (the *requeue target*). The profile of the entry, or the profile or prefixed profile of the procedure, shall either have no parameters, or be type conformant (see 6.3.1) with the profile of the innermost enclosing entry_body or accept_statement.

Legality Rules

4 A requeue_statement shall be within a callable construct that is either an entry_body or an accept_statement, and this construct shall be the innermost enclosing body or callable construct.

5/3 {*AI05-0030-2*} {*AI05-0215-1*} If the requeue target has parameters, then its (prefixed) profile shall be subtype conformant with the profile of the innermost enclosing callable construct.

5.1/4 {*AI12-0090-1*} Given a requeue_statement where the innermost enclosing callable construct is for an entry *E1*, for every [specific or class-wide]postcondition expression *P1* that applies to *E1*, there shall exist a postcondition expression *P2* that applies to the requeue target *E2* such that

5.2/4 - *P1* is fully conformant with the expression produced by replacing each reference in *P2* to a formal parameter of *E2* with a reference to the corresponding formal paramter of *E1*; and

5.3/4 - if *P1* is enabled, then *P2* is also enabled.

5.a/4 **Discussion:** Roughly speaking, the postcondition of the requeue target is required to imply that of the enclosing callable construct.

5.4/5 {*AI12-0090-1*} {*AI12-0143-1*} The requeue target shall not have an applicable specific or class-wide postcondition that includes an Old or Index attribute_reference.

{*AI12-0090-1*} If the requeue target is declared immediately within the task_definition of a named task type or the protected_definition of a named protected type, and if the requeue statement occurs within the body of that type, and if the requeue is an external requeue, then the requeue target shall not have a specific or class-wide postcondition which includes a name denoting either the current instance of that type or any entity declared within the declaration of that type. 5.5/4

Discussion: The above pair of rules always apply; they don't depend on whether or not any of the postconditions are enabled. 5.b/4

{*AI05-0030-2*} {*AI05-0215-1*} {*AI12-0090-1*} If the target is a procedure, the name shall denote a renaming of an entry, or shall denote a view or a prefixed view of a primitive subprogram of a synchronized interface, where the first parameter of the unprefixed view of the primitive subprogram shall be a controlling parameter, and the Synchronization aspect shall be specified with synchronization_kind By_Entry for the primitive subprogram. 5.6/4

{*AI05-0030-2*} In a requeue_statement of an accept_statement of some task unit, either the target object shall be a part of a formal parameter of the accept_statement, or the accessibility level of the target object shall not be equal to or statically deeper than any enclosing accept_statement of the task unit. In a requeue_statement of an entry_body of some protected unit, either the target object shall be a part of a formal parameter of the entry_body, or the accessibility level of the target object shall not be statically deeper than that of the entry_declaration for the entry_body. 6/3

Ramification: In the entry_body case, the intent is that the target object can be global, or can be a component of the protected unit, but cannot be a local variable of the entry_body. 6.a

Reason: These restrictions ensure that the target object of the requeue outlives the completion and finalization of the enclosing callable construct. They also prevent requeuing from a nested accept_statement on a parameter of an outer accept_statement, which could create some strange "long-distance" connections between an entry caller and its server. 6.b

Note that in the strange case where a task_body is nested inside an accept_statement, it is permissible to requeue from an accept_statement of the inner task_body on parameters of the outer accept_statement. This is not a problem because all calls on the inner task have to complete before returning from the outer accept_statement, meaning no "dangling calls" will be created. 6.c

Implementation Note: By disallowing certain requeues, we ensure that the normal terminate_alternative rules remain sensible, and that explicit clearing of the entry queues of a protected object during finalization is rarely necessary. In particular, such clearing of the entry queues is necessary only (ignoring premature Unchecked_Deallocation) for protected objects declared in a task_body (or created by an allocator for an access type declared in such a body) containing one or more requeue_statements. Protected objects declared in subprograms, or at the library level, will never need to have their entry queues explicitly cleared during finalization. 6.d

Dynamic Semantics

{*AI05-0030-2*} {*AI12-0090-1*} {*AI12-0335-1*} The execution of a requeue_statement begins with the following sequence of steps: 7/5

1. The *procedure_or_entry_*name is evaluated. This includes evaluation of the prefix (if any) identifying the target task or protected object and of the expression (if any) identifying the entry within an entry family. 7.1/5
2. {*AI12-0335-1*} If the target object is not a part of a formal parameter of the innermost enclosing callable construct, a check is made that the accessibility level of the target object is not equal to or deeper than the level of the innermost enclosing callable construct. If this check fails, Program_Error is raised. 7.2/5
3. {*AI12-0090-1*} Precondition checks are performed as for a call to the requeue target. 7.3/5
4. The entry_body or accept_statement enclosing the requeue_statement is then completed[, finalized, and left (see 7.6.1)]. 7.4/5

8 For the execution of a requeue on an entry of a target task, after leaving the enclosing callable construct, the named entry is checked to see if it is open and the requeued call is either selected immediately or queued, as for a normal entry call (see 9.5.3).

9 For the execution of a requeue on an entry of a target protected object, after leaving the enclosing callable construct:

10 • if the requeue is an internal requeue (that is, the requeue is back on an entry of the same protected object — see 9.5), the call is added to the queue of the named entry and the ongoing protected action continues (see 9.5.1);

10.a **Ramification:** Note that for an internal requeue, the call is queued without checking whether the target entry is open. This is because the entry queues will be serviced before the current protected action completes anyway, and considering the requeued call immediately might allow it to "jump" ahead of existing callers on the same queue.

11 • if the requeue is an external requeue (that is, the target protected object is not implicitly the same as the current object — see 9.5), a protected action is started on the target object and proceeds as for a normal entry call (see 9.5.3).

12/4 {*AI05-0030-2*} {*AI05-0090-1*} If the requeue target named in the requeue_statement has formal parameters, then during the execution of the accept_statement or entry_body corresponding to the new entry and during the checking of any preconditions of the new entry, the formal parameters denote the same objects as did the corresponding formal parameters of the callable construct completed by the requeue. [In any case, no parameters are specified in a requeue_statement; any parameter passing is implicit.]

13 If the requeue_statement includes the reserved words **with abort** (it is a *requeue-with-abort*), then:

14 • if the original entry call has been aborted (see 9.8), then the requeue acts as an abort completion point for the call, and the call is cancelled and no requeue is performed;

15 • if the original entry call was timed (or conditional), then the original expiration time is the expiration time for the requeued call.

16 If the reserved words **with abort** do not appear, then the call remains protected against cancellation while queued as the result of the requeue_statement.

16.a **Ramification:** This protection against cancellation lasts only until the call completes or a subsequent requeue-with-abort is performed on the call.

16.b **Reason:** We chose to protect a requeue, by default, against abort or cancellation. This seemed safer, since it is likely that extra steps need to be taken to allow for possible cancellation once the servicing of an entry call has begun. This also means that in the absence of **with abort** the usual Ada 83 behavior is preserved, namely that once an entry call is accepted, it cannot be cancelled until it completes.

17/5 NOTE {*AI12-0300-1*} A requeue is permitted from a single entry to an entry of an entry family, or vice versa. The entry index, if any, plays no part in the subtype conformance check between the profiles of the two entries; an entry index is part of the *entry*_name for an entry of a family.

Examples

18 *Examples of requeue statements:*

19
```
requeue Request(Medium) with abort;
                        -- requeue on a member of an entry family of the current task, see 9.1
```

20
```
requeue Flags(I).Seize;
                        -- requeue on an entry of an array component, see 9.4
```

Extensions to Ada 83

20.a The requeue_statement is new.

Extensions to Ada 2005

{*AI05-0030-2*} {*AI05-0215-1*} Added the ability to requeue on operations of synchronized interfaces that are declared to be an entry. 20.b/3

Inconsistencies With Ada 2012

{*AI12-0090-1*} **Corrigendum:** We now define that any preconditions of the requeue target are evaluated as part of a requeue_statement. Original Ada 2012 did not specify this, so a program that requeues when the preconditions fail will raise an exception when none would happen in original Ada 2012. We don't expect this to be a problem, as in that case, the entry body would be called with some of its preconditions evaluating as False; the body is likely to assume that they are true and probably will have failed in some other way anyway. 20.c/4

{*AI12-0335-1*} **Correction:** We now include an accessibility check on requeues. This means Program_Error could be raised for a requeue that worked in Ada 2012. This can only fail for an object for which the statically deeper relationship does not apply, for instance a stand-alone object of an anonymous access type. Most programs that are affected are erroneous anyway (as they will eventually use a nonexistent object), so we do not believe this will matter in practice. 20.d/5

Incompatibilities With Ada 2012

{*AI12-0090-1*} **Corrigendum:** If a requeue target has a different postcondition than the original entry, the requeue is now illegal. In such a case, the original postcondition would never have been evaluated, and assumptions that the caller relied upon might not be true. A requeue should be invisible to the caller with respect to any postconditions; thus we only allow it when the original entry has no postconditions or the requeue target has (at least) the same postconditions. 20.e/4

Wording Changes from Ada 2012

{*AI12-0143-1*} Added a Legality Rule for the new Index attribute (see 6.1.1). 20.f/5

9.6 Delay Statements, Duration, and Time

[A delay_statement is used to block further execution until a specified *expiration time* is reached. The expiration time can be specified either as a particular point in time (in a delay_until_statement), or in seconds from the current time (in a delay_relative_statement). The language-defined package Calendar provides definitions for a type Time and associated operations, including a function Clock that returns the current time.] 1

Syntax

delay_statement ::= delay_until_statement | delay_relative_statement 2

delay_until_statement ::= **delay until** *delay_*expression; 3

delay_relative_statement ::= **delay** *delay_*expression; 4

Name Resolution Rules

The expected type for the *delay_*expression in a delay_relative_statement is the predefined type Duration. The *delay_*expression in a delay_until_statement is expected to be of any nonlimited type. 5

Legality Rules

{*AI05-0092-1*} There can be multiple time bases, each with a corresponding clock, and a corresponding *time type*. The type of the *delay_*expression in a delay_until_statement shall be a time type — either the type Time defined in the language-defined package Calendar (see below), the type Time in the package Real_Time (see D.8), or some other implementation-defined time type. 6/3

Implementation defined: Any implementation-defined time types. 6.a

Static Semantics

7 [There is a predefined fixed point type named Duration, declared in the visible part of package Standard;] a value of type Duration is used to represent the length of an interval of time, expressed in seconds. [The type Duration is not specific to a particular time base, but can be used with any time base.]

8/3 {*AI05-0092-1*} A value of the type Time in package Calendar, or of some other time type, represents a time as reported by a corresponding clock.

9 The following language-defined library package exists:

10/5
```
{AI12-0241-1} {AI12-0302-1}
package Ada.Calendar
  with Nonblocking, Global => in out synchronized is
  type Time is private;

11/2  {AI95-00351-01}   subtype Year_Number  is Integer range 1901 .. 2399;
  subtype Month_Number is Integer range 1 .. 12;
  subtype Day_Number   is Integer range 1 .. 31;
  subtype Day_Duration is Duration range 0.0 .. 86_400.0;
```

11.a/2 **Reason:** {*AI95-00351-01*} A range of 500 years was chosen, as that only requires one extra bit for the year as compared to Ada 95. This was done to minimize disruptions with existing implementations. (One implementor reports that their time values represent nanoseconds, and this year range requires 63.77 bits to represent.)

12
```
  function Clock return Time;
```

13
```
  function Year   (Date : Time) return Year_Number;
  function Month  (Date : Time) return Month_Number;
  function Day    (Date : Time) return Day_Number;
  function Seconds(Date : Time) return Day_Duration;
```

14
```
  procedure Split (Date  : in Time;
                   Year    : out Year_Number;
                   Month   : out Month_Number;
                   Day     : out Day_Number;
                   Seconds : out Day_Duration);
```

15
```
  function Time_Of(Year  : Year_Number;
                   Month   : Month_Number;
                   Day     : Day_Number;
                   Seconds : Day_Duration := 0.0)
    return Time;
```

16
```
  function "+" (Left : Time;   Right : Duration) return Time;
  function "+" (Left : Duration; Right : Time) return Time;
  function "-" (Left : Time;   Right : Duration) return Time;
  function "-" (Left : Time;   Right : Time) return Duration;
```

17
```
  function "<" (Left, Right : Time) return Boolean;
  function "<="(Left, Right : Time) return Boolean;
  function ">" (Left, Right : Time) return Boolean;
  function ">="(Left, Right : Time) return Boolean;
```

18
```
  Time_Error : exception;
```

19
```
private
   ... -- not specified by the language
end Ada.Calendar;
```

Dynamic Semantics

20 For the execution of a delay_statement, the *delay*_expression is first evaluated. For a delay_until_statement, the expiration time for the delay is the value of the *delay*_expression, in the time base associated with the type of the expression. For a delay_relative_statement, the expiration time is defined as the current time, in the time base associated with relative delays, plus the value of the *delay*_expression converted to the type Duration, and then rounded up to the next clock tick. The time base associated with relative delays is as defined in D.9, “Delay Accuracy” or is implementation defined.

20.a **Implementation defined:** The time base associated with relative delays.

Ramification: Rounding up to the next clock tick means that the reading of the delay-relative clock when the delay expires should be no less than the current reading of the delay-relative clock plus the specified duration. 20.b

The task executing a delay_statement is blocked until the expiration time is reached, at which point it becomes ready again. If the expiration time has already passed, the task is not blocked. 21

Discussion: For a delay_relative_statement, this case corresponds to when the value of the *delay*_expression is zero or negative. 21.a

Even though the task is not blocked, it might be put back on the end of its ready queue. See D.2, “Priority Scheduling”. 21.b

{*AI05-0092-1*} If an attempt is made to *cancel* the delay_statement [(as part of an asynchronous_select or abort — see 9.7.4 and 9.8)], the statement is cancelled if the expiration time has not yet passed, thereby completing the delay_statement. 22/3

Reason: This is worded this way so that in an asynchronous_select where the triggering_statement is a delay_statement, an attempt to cancel the delay when the abortable_part completes is ignored if the expiration time has already passed, in which case the optional statements of the triggering_alternative are executed. 22.a

The time base associated with the type Time of package Calendar is implementation defined. The function Clock of package Calendar returns a value representing the current time for this time base. [The implementation-defined value of the named number System.Tick (see 13.7) is an approximation of the length of the real-time interval during which the value of Calendar.Clock remains constant.] 23

Implementation defined: The time base of the type Calendar.Time. 23.a

{*AI95-00351-01*} The functions Year, Month, Day, and Seconds return the corresponding values for a given value of the type Time, as appropriate to an implementation-defined time zone; the procedure Split returns all four corresponding values. Conversely, the function Time_Of combines a year number, a month number, a day number, and a duration, into a value of type Time. The operators "+" and "–" for addition and subtraction of times and durations, and the relational operators for times, have the conventional meaning. 24/2

Implementation defined: The time zone used for package Calendar operations. 24.a/2

Ramification: {*AI05-0119-1*} The behavior of these values and subprograms if the time zone changes is also implementation-defined. In particular, the changes associated with summer time adjustments (like Daylight Savings Time in the United States) should be treated as a change in the implementation-defined time zone. The language does not specify whether the time zone information is stored in values of type Time; therefore the results of binary operators are unspecified when the operands are the two values with different effective time zones. In particular, the results of "-" may differ from the "real" result by the difference in the time zone adjustment. Similarly, the result of UTC_Time_Offset (see 9.6.1) may or may not reflect a time zone adjustment. 24.b/3

If Time_Of is called with a seconds value of 86_400.0, the value returned is equal to the value of Time_Of for the next day with a seconds value of 0.0. The value returned by the function Seconds or through the Seconds parameter of the procedure Split is always less than 86_400.0. 25

{*8652/0030*} {*AI95-00113-01*} The exception Time_Error is raised by the function Time_Of if the actual parameters do not form a proper date. This exception is also raised by the operators "+" and "–" if the result is not representable in the type Time or Duration, as appropriate. This exception is also raised by the functions Year, Month, Day, and Seconds and the procedure Split if the year number of the given date is outside of the range of the subtype Year_Number. 26/1

To be honest: {*8652/0106*} {*AI95-00160-01*} By "proper date" above we mean that the given year has a month with the given day. For example, February 29th is a proper date only for a leap year. We do not mean to include the Seconds in this notion; in particular, we do not mean to require implementations to check for the “missing hour” that occurs when Daylight Savings Time starts in the spring. 26.a/1

Reason: {*8652/0030*} {*AI95-00113-01*} {*AI95-00351-01*} We allow Year and Split to raise Time_Error because the arithmetic operators are allowed (but not required) to produce times that are outside the range of years from 1901 to 2399. This is similar to the way integer operators may return values outside the base range of their type so long as the value is mathematically correct. We allow the functions Month, Day and Seconds to raise Time_Error so that they can be implemented in terms of Split. 26.b/2

Implementation Requirements

27 The implementation of the type Duration shall allow representation of time intervals (both positive and negative) up to at least 86400 seconds (one day); Duration'Small shall not be greater than twenty milliseconds. The implementation of the type Time shall allow representation of all dates with year numbers in the range of Year_Number[; it may allow representation of other dates as well (both earlier and later).]

Implementation Permissions

28/3 {*AI05-0092-1*} An implementation may define additional time types.

29 An implementation may raise Time_Error if the value of a *delay*_expression in a delay_until_statement of a select_statement represents a time more than 90 days past the current time. The actual limit, if any, is implementation-defined.

29.a **Implementation defined:** Any limit on delay_until_statements of select_statements.

29.b **Implementation Note:** This allows an implementation to implement select_statement timeouts using a representation that does not support the full range of a time type. In particular 90 days of seconds can be represented in 23 bits, allowing a signed 24-bit representation for the seconds part of a timeout. There is no similar restriction allowed for stand-alone delay_until_statements, as these can be implemented internally using a loop if necessary to accommodate a long delay.

Implementation Advice

30 Whenever possible in an implementation, the value of Duration'Small should be no greater than 100 microseconds.

30.a **Implementation Note:** This can be satisfied using a 32-bit 2's complement representation with a *small* of 2.0**(–14) — that is, 61 microseconds — and a range of ± 2.0**17 — that is, 131_072.0.

30.b/2 **Implementation Advice:** The value of Duration'Small should be no greater than 100 microseconds.

31/5 {*AI12-0444-1*} The time base for delay_relative_statements should be monotonic; it can be different than the time base as used for Calendar.Clock.

31.a/2 **Implementation Advice:** The time base for delay_relative_statements should be monotonic.

32 NOTE 1 A delay_relative_statement with a negative value of the *delay*_expression is equivalent to one with a zero value.

33/5 NOTE 2 {*AI12-0440-1*} A delay_statement can be executed by the environment task; consequently delay_statements can be executed as part of the elaboration of a library_item or the execution of the main subprogram. Such statements delay the environment task (see 10.2).

34 NOTE 3 A delay_statement is an abort completion point and a potentially blocking operation, even if the task is not actually blocked.

35 NOTE 4 There is no necessary relationship between System.Tick (the resolution of the clock of package Calendar) and Duration'Small (the *small* of type Duration).

35.a **Ramification:** The inaccuracy of the delay_statement has no relation to System.Tick. In particular, it is possible that the clock used for the delay_statement is less accurate than Calendar.Clock.

35.b We considered making Tick a run-time-determined quantity, to allow for easier configurability. However, this would not be upward compatible, and the desired configurability can be achieved using functionality defined in Annex D, “Real-Time Systems”.

36 NOTE 5 Additional requirements associated with delay_statements are given in D.9, “Delay Accuracy”.

Examples

37 *Example of a relative delay statement:*

38
```
delay 3.0;  -- delay 3.0 seconds
```

Example of a periodic task: 39

```
declare                                                                    40
   use Ada.Calendar;
   Next_Time : Time := Clock + Period;
                       -- Period is a global constant of type Duration
begin
   loop                -- repeated every Period seconds
      delay until Next_Time;
      ... -- perform some actions
      Next_Time := Next_Time + Period;
   end loop;
end;
```

Inconsistencies With Ada 83

For programs that raise Time_Error on "+" or "–" in Ada 83,the exception might be deferred until a call on Split or Year_Number, or might not be raised at all (if the offending time is never Split after being calculated). This should not affect typical programs, since they deal only with times corresponding to the relatively recent past or near future. 40.a

Extensions to Ada 83

The syntax rule for delay_statement is modified to allow delay_until_statements. 40.b

{*AI95-00351-01*} The type Time may represent dates with year numbers outside of Year_Number. Therefore, the operations "+" and "–" need only raise Time_Error if the result is not representable in Time (or Duration); also, Split or Year will now raise Time_Error if the year number is outside of Year_Number. This change is intended to simplify the implementation of "+" and "–" (allowing them to depend on overflow for detecting when to raise Time_Error) and to allow local time zone information to be considered at the time of Split rather than Clock (depending on the implementation approach). For example, in a POSIX environment, it is natural for the type Time to be based on GMT, and the results of procedure Split (and the functions Year, Month, Day, and Seconds) to depend on local time zone information. In other environments, it is more natural for the type Time to be based on the local time zone, with the results of Year, Month, Day, and Seconds being pure functions of their input. 40.c/2

This paragraph was deleted.{*AI95-00351-01*} 40.d/2

Inconsistencies With Ada 95

{*AI95-00351-01*} The upper bound of Year_Number has been changed to avoid a year 2100 problem. A program which expects years past 2099 to raise Constraint_Error will fail in Ada 2005. We don't expect there to be many programs which are depending on an exception to be raised. A program that uses Year_Number'Last as a magic number may also fail if values of Time are stored outside of the program. Note that the lower bound of Year_Number wasn't changed, because it is not unusual to use that value in a constant to represent an unknown time. 40.e/2

Wording Changes from Ada 95

{*8652/0002*} {*AI95-00171-01*} **Corrigendum:** Clarified that Month, Day, and Seconds can raise Time_Error. 40.f/2

9.6.1 Formatting, Time Zones, and other operations for Time

Static Semantics

{*AI95-00351-01*} {*AI95-00427-01*} The following language-defined library packages exist: 1/2

```
{AI12-0241-1} {AI12-0302-1} package Ada.Calendar.Time_Zones                2/5
   with Nonblocking, Global => in out synchronized is
```

```
   -- Time zone manipulation:                                              3/2
```

```
   type Time_Offset is range -28*60 .. 28*60;                              4/2
```

Reason: {*AI12-0005-1*} We want to be able to specify the difference between any two arbitrary time zones. You might think that 1440 (24 hours) would be enough, but there are places (like Tonga, which is UTC+13hr) which are more than 12 hours different than UTC. Combined with summer time (known as daylight saving time in some parts of the world) – which switches opposite in the northern and southern hemispheres – and even greater differences are possible. We know of cases of a 26 hours difference, so we err on the safe side by selecting 28 hours as the limit. 4.a/4

```
   Unknown_Zone_Error : exception;                                         5/2
```

6/5 {*AI12-0336-1*} **function** Local_Time_Offset (Date : Time := Clock) **return** Time_Offset;

6.1/5 {*AI12-0336-1*} **function** UTC_Time_Offset (Date : Time := Clock) **return** Time_Offset **renames** Local_Time_Offset;

7/2 **end** Ada.Calendar.Time_Zones;

8/5 {*AI12-0241-1*} {*AI12-0302-1*}
package Ada.Calendar.Arithmetic
with Nonblocking, Global => **in out synchronized** **is**

9/2 -- *Arithmetic on days:*

10/2 **type** Day_Count **is range**
-366*(1+Year_Number'Last - Year_Number'First)
..
366*(1+Year_Number'Last - Year_Number'First);

11/2 **subtype** Leap_Seconds_Count **is** Integer **range** -2047 .. 2047;

11.a/2 **Reason:** The maximum number of leap seconds is likely to be much less than this, but we don't want to reach the limit too soon if the earth's behavior suddenly changes. We believe that the maximum number is 1612, based on the current rules, but that number is too weird to use here.

12/2 **procedure** Difference (Left, Right : **in** Time;
Days : **out** Day_Count;
Seconds : **out** Duration;
Leap_Seconds : **out** Leap_Seconds_Count);

13/2 **function** "+" (Left : Time; Right : Day_Count) **return** Time;
function "+" (Left : Day_Count; Right : Time) **return** Time;
function "-" (Left : Time; Right : Day_Count) **return** Time;
function "-" (Left, Right : Time) **return** Day_Count;

14/2 **end** Ada.Calendar.Arithmetic;

15/5 {*AI12-0241-1*} {*AI12-0302-1*}
with Ada.Calendar.Time_Zones;
package Ada.Calendar.Formatting
with Nonblocking, Global => **in out synchronized** **is**

16/2 -- *Day of the week:*

17/2 **type** Day_Name **is** (Monday, Tuesday, Wednesday, Thursday,
Friday, Saturday, Sunday);

18/2 **function** Day_of_Week (Date : Time) **return** Day_Name;

19/2 -- *Hours:Minutes:Seconds access:*

20/2 **subtype** Hour_Number **is** Natural **range** 0 .. 23;
subtype Minute_Number **is** Natural **range** 0 .. 59;
subtype Second_Number **is** Natural **range** 0 .. 59;
subtype Second_Duration **is** Day_Duration **range** 0.0 .. 1.0;

21/2 **function** Year (Date : Time;
Time_Zone : Time_Zones.Time_Offset := 0)
return Year_Number;

22/2 **function** Month (Date : Time;
Time_Zone : Time_Zones.Time_Offset := 0)
return Month_Number;

23/2 **function** Day (Date : Time;
Time_Zone : Time_Zones.Time_Offset := 0)
return Day_Number;

24/2 **function** Hour (Date : Time;
Time_Zone : Time_Zones.Time_Offset := 0)
return Hour_Number;

25/2 **function** Minute (Date : Time;
Time_Zone : Time_Zones.Time_Offset := 0)
return Minute_Number;

26/2 **function** Second (Date : Time)
return Second_Number;

```
function Sub_Second (Date : Time)                                                    27/2
                        return Second_Duration;

function Seconds_Of (Hour   :  Hour_Number;                                          28/2
                     Minute : Minute_Number;
                     Second : Second_Number := 0;
                     Sub_Second : Second_Duration := 0.0)
                        return Day_Duration;

procedure Split (Seconds    : in Day_Duration;                                       29/2
                 Hour       : out Hour_Number;
                 Minute     : out Minute_Number;
                 Second     : out Second_Number;
                 Sub_Second : out Second_Duration);

function Time_Of (Year       : Year_Number;                                          30/2
                  Month      : Month_Number;
                  Day        : Day_Number;
                  Hour       : Hour_Number;
                  Minute     : Minute_Number;
                  Second     : Second_Number;
                  Sub_Second : Second_Duration := 0.0;
                  Leap_Second: Boolean := False;
                  Time_Zone  : Time_Zones.Time_Offset := 0)
                        return Time;

function Time_Of (Year       : Year_Number;                                          31/2
                  Month      : Month_Number;
                  Day        : Day_Number;
                  Seconds    : Day_Duration := 0.0;
                  Leap_Second: Boolean := False;
                  Time_Zone  : Time_Zones.Time_Offset := 0)
                        return Time;

procedure Split (Date       : in Time;                                               32/2
                 Year       : out Year_Number;
                 Month      : out Month_Number;
                 Day        : out Day_Number;
                 Hour       : out Hour_Number;
                 Minute     : out Minute_Number;
                 Second     : out Second_Number;
                 Sub_Second : out Second_Duration;
                 Time_Zone  : in Time_Zones.Time_Offset := 0);

procedure Split (Date       : in Time;                                               33/2
                 Year       : out Year_Number;
                 Month      : out Month_Number;
                 Day        : out Day_Number;
                 Hour       : out Hour_Number;
                 Minute     : out Minute_Number;
                 Second     : out Second_Number;
                 Sub_Second : out Second_Duration;
                 Leap_Second: out Boolean;
                 Time_Zone  : in Time_Zones.Time_Offset := 0);

procedure Split (Date       : in Time;                                               34/2
                 Year       : out Year_Number;
                 Month      : out Month_Number;
                 Day        : out Day_Number;
                 Seconds    : out Day_Duration;
                 Leap_Second: out Boolean;
                 Time_Zone  : in Time_Zones.Time_Offset := 0);

-- Simple image and value:                                                           35/2
function Image (Date : Time;
                Include_Time_Fraction : Boolean := False;
                Time_Zone  : Time_Zones.Time_Offset := 0) return String;
```

35.1/5
```
{AI12-0336-1} {AI12-0347-1}    function Local_Image (Date : Time;
                              Include_Time_Fraction : Boolean := False)
      return String is
      (Image (Date, Include_Time_Fraction,
              Time_Zones.Local_Time_Offset (Date)));
```

36/2
```
   function Value (Date : String;
                   Time_Zone  : Time_Zones.Time_Offset := 0) return Time;
```

37/2
```
   function Image (Elapsed_Time : Duration;
                   Include_Time_Fraction : Boolean := False) return String;
```

38/2
```
   function Value (Elapsed_Time : String) return Duration;
```

39/2
```
end Ada.Calendar.Formatting;
```

40/5 {*AI95-00351-01*} {*AI12-0336-1*} Type Time_Offset represents for a given locality at a given moment the number of minutes the local time is, at that moment, ahead (+) or behind (-) Coordinated Universal Time (abbreviated UTC).[The Time_Offset for UTC is zero].

41/5
```
{AI12-0336-1} function Local_Time_Offset (Date : Time := Clock) return Time_Offset;
```

42/5 {*AI95-00351-01*} {*AI05-0119-1*} {*AI05-0269-1*} {*AI12-0336-1*} Returns, as a number of minutes, the Time_Offset of the implementation-defined time zone of Calendar, at the time Date. If the time zone of the Calendar implementation is unknown, then Unknown_Zone_Error is raised.

42.a.1/3 **Ramification:** {*AI05-0119-1*} In North America, the result will be negative; in Europe, the result will be zero or positive.

42.a/5 **Discussion:** {*AI12-0336-1*} The Date parameter is needed to take into account time differences caused by daylight-savings time and other time changes.

42.b/2 Other time zones can be supported with a child package. We don't define one because of the lack of agreement on the definition of a time zone.

42.c/2 The accuracy of this routine is not specified; the intent is that the facilities of the underlying target operating system are used to implement it.

43/2
```
procedure Difference (Left, Right : in Time;
                      Days : out Day_Count;
                      Seconds : out Duration;
                      Leap_Seconds : out Leap_Seconds_Count);
```

44/2 {*AI95-00351-01*} {*AI95-00427-01*} Returns the difference between Left and Right. Days is the number of days of difference, Seconds is the remainder seconds of difference excluding leap seconds, and Leap_Seconds is the number of leap seconds. If Left < Right, then Seconds <= 0.0, Days <= 0, and Leap_Seconds <= 0. Otherwise, all values are nonnegative. The absolute value of Seconds is always less than 86_400.0. For the returned values, if Days = 0, then Seconds + Duration(Leap_Seconds) = Calendar."–" (Left, Right).

44.a/2 **Discussion:** Leap_Seconds, if any, are not included in Seconds. However, Leap_Seconds should be included in calculations using the operators defined in Calendar, as is specified for "–" above.

45/2
```
function "+" (Left : Time; Right : Day_Count) return Time;
function "+" (Left : Day_Count; Right : Time) return Time;
```

46/2 {*AI95-00351-01*} Adds a number of days to a time value. Time_Error is raised if the result is not representable as a value of type Time.

47/2
```
function "-" (Left : Time; Right : Day_Count) return Time;
```

48/2 {*AI95-00351-01*} Subtracts a number of days from a time value. Time_Error is raised if the result is not representable as a value of type Time.

```
function "-" (Left, Right : Time) return Day_Count;
```
49/2

{*AI95-00351-01*} Subtracts two time values, and returns the number of days between them. This is the same value that Difference would return in Days. 50/2

```
function Day_of_Week (Date : Time) return Day_Name;
```
51/2

{*AI95-00351-01*} Returns the day of the week for Time. This is based on the Year, Month, and Day values of Time. 52/2

```
function Year       (Date : Time;
                     Time_Zone  : Time_Zones.Time_Offset := 0)
                        return Year_Number;
```
53/2

{*AI95-00427-01*} Returns the year for Date, as appropriate for the specified time zone offset. 54/2

```
function Month      (Date : Time;
                     Time_Zone  : Time_Zones.Time_Offset := 0)
                        return Month_Number;
```
55/2

{*AI95-00427-01*} Returns the month for Date, as appropriate for the specified time zone offset. 56/2

```
function Day        (Date : Time;
                     Time_Zone  : Time_Zones.Time_Offset := 0)
                        return Day_Number;
```
57/2

{*AI95-00427-01*} Returns the day number for Date, as appropriate for the specified time zone offset. 58/2

```
function Hour       (Date : Time;
                     Time_Zone  : Time_Zones.Time_Offset := 0)
                        return Hour_Number;
```
59/2

{*AI95-00351-01*} Returns the hour for Date, as appropriate for the specified time zone offset. 60/2

```
function Minute     (Date : Time;
                     Time_Zone  : Time_Zones.Time_Offset := 0)
                        return Minute_Number;
```
61/2

{*AI95-00351-01*} Returns the minute within the hour for Date, as appropriate for the specified time zone offset. 62/2

```
function Second     (Date : Time)
                        return Second_Number;
```
63/2

{*AI95-00351-01*} {*AI95-00427-01*} Returns the second within the hour and minute for Date. 64/2

```
function Sub_Second (Date : Time)
                        return Second_Duration;
```
65/2

{*AI95-00351-01*} {*AI95-00427-01*} Returns the fraction of second for Date (this has the same accuracy as Day_Duration). The value returned is always less than 1.0. 66/2

```
function Seconds_Of (Hour   : Hour_Number;
                     Minute : Minute_Number;
                     Second : Second_Number := 0;
                     Sub_Second : Second_Duration := 0.0)
                        return Day_Duration;
```
67/2

{*AI95-00351-01*} {*AI95-00427-01*} Returns a Day_Duration value for the combination of the given Hour, Minute, Second, and Sub_Second. This value can be used in Calendar.Time_Of as well as the argument to Calendar."+" and Calendar."–". If Seconds_Of is called with a Sub_Second value of 1.0, the value returned is equal to the value of Seconds_Of for the next second with a Sub_Second value of 0.0. 68/2

69/2
```
procedure Split (Seconds    : in Day_Duration;
                 Hour       : out Hour_Number;
                 Minute     : out Minute_Number;
                 Second     : out Second_Number;
                 Sub_Second : out Second_Duration);
```

70/3 {*AI95-00351-01*} {*AI95-00427-01*} {*AI05-0238-1*} Splits Seconds into Hour, Minute, Second and Sub_Second in such a way that the resulting values all belong to their respective subtypes. The value returned in the Sub_Second parameter is always less than 1.0. If Seconds = 86400.0, Split propagates Time_Error.

70.a/2 **Ramification:** There is only one way to do the split which meets all of the requirements.

70.b/3 **Reason:** {*AI05-0238-1*} If Seconds = 86400.0, one of the returned values would have to be out of its defined range (either Sub_Second = 1.0 or Hour = 24 with the other value being 0). This doesn't seem worth breaking the invariants.

71/2
```
function Time_Of (Year       : Year_Number;
                  Month      : Month_Number;
                  Day        : Day_Number;
                  Hour       : Hour_Number;
                  Minute     : Minute_Number;
                  Second     : Second_Number;
                  Sub_Second : Second_Duration := 0.0;
                  Leap_Second: Boolean := False;
                  Time_Zone  : Time_Zones.Time_Offset := 0)
                  return Time;
```

72/2 {*AI95-00351-01*} {*AI95-00427-01*} If Leap_Second is False, returns a Time built from the date and time values, relative to the specified time zone offset. If Leap_Second is True, returns the Time that represents the time within the leap second that is one second later than the time specified by the other parameters. Time_Error is raised if the parameters do not form a proper date or time. If Time_Of is called with a Sub_Second value of 1.0, the value returned is equal to the value of Time_Of for the next second with a Sub_Second value of 0.0.

72.a/2 **Discussion:** Time_Error should be raised if Leap_Second is True, and the date and time values do not represent the second before a leap second. A leap second always occurs at midnight UTC, and is 23:59:60 UTC in ISO notation. So, if the time zone is UTC and Leap_Second is True, if any of Hour /= 23, Minute /= 59, or Second /= 59, then Time_Error should be raised. However, we do not say that, because other time zones will have different values where a leap second is allowed.

73/2
```
function Time_Of (Year       : Year_Number;
                  Month      : Month_Number;
                  Day        : Day_Number;
                  Seconds    : Day_Duration := 0.0;
                  Leap_Second: Boolean := False;
                  Time_Zone  : Time_Zones.Time_Offset := 0)
                  return Time;
```

74/2 {*AI95-00351-01*} {*AI95-00427-01*} If Leap_Second is False, returns a Time built from the date and time values, relative to the specified time zone offset. If Leap_Second is True, returns the Time that represents the time within the leap second that is one second later than the time specified by the other parameters. Time_Error is raised if the parameters do not form a proper date or time. If Time_Of is called with a Seconds value of 86_400.0, the value returned is equal to the value of Time_Of for the next day with a Seconds value of 0.0.

```
procedure Split (Date       : in Time;                                    75/2
                 Year       : out Year_Number;
                 Month      : out Month_Number;
                 Day        : out Day_Number;
                 Hour       : out Hour_Number;
                 Minute     : out Minute_Number;
                 Second     : out Second_Number;
                 Sub_Second : out Second_Duration;
                 Leap_Second: out Boolean;
                 Time_Zone  : in Time_Zones.Time_Offset := 0);
```

{*AI95-00351-01*} {*AI95-00427-01*} If Date does not represent a time within a leap second, splits Date into its constituent parts (Year, Month, Day, Hour, Minute, Second, Sub_Second), relative to the specified time zone offset, and sets Leap_Second to False. If Date represents a time within a leap second, set the constituent parts to values corresponding to a time one second earlier than that given by Date, relative to the specified time zone offset, and sets Leap_Seconds to True. The value returned in the Sub_Second parameter is always less than 1.0. 76/2

```
procedure Split (Date       : in Time;                                    77/2
                 Year       : out Year_Number;
                 Month      : out Month_Number;
                 Day        : out Day_Number;
                 Hour       : out Hour_Number;
                 Minute     : out Minute_Number;
                 Second     : out Second_Number;
                 Sub_Second : out Second_Duration;
                 Time_Zone  : in Time_Zones.Time_Offset := 0);
```

{*AI95-00351-01*} {*AI95-00427-01*} Splits Date into its constituent parts (Year, Month, Day, Hour, Minute, Second, Sub_Second), relative to the specified time zone offset. The value returned in the Sub_Second parameter is always less than 1.0. 78/2

```
procedure Split (Date       : in Time;                                    79/2
                 Year       : out Year_Number;
                 Month      : out Month_Number;
                 Day        : out Day_Number;
                 Seconds    : out Day_Duration;
                 Leap_Second: out Boolean;
                 Time_Zone  : in Time_Zones.Time_Offset := 0);
```

{*AI95-00351-01*} {*AI95-00427-01*} If Date does not represent a time within a leap second, splits Date into its constituent parts (Year, Month, Day, Seconds), relative to the specified time zone offset, and sets Leap_Second to False. If Date represents a time within a leap second, set the constituent parts to values corresponding to a time one second earlier than that given by Date, relative to the specified time zone offset, and sets Leap_Seconds to True. The value returned in the Seconds parameter is always less than 86_400.0. 80/2

```
function Image (Date : Time;                                              81/2
                Include_Time_Fraction : Boolean := False;
                Time_Zone  : Time_Zones.Time_Offset := 0) return String;
```

{*AI95-00351-01*} Returns a string form of the Date relative to the given Time_Zone. The format is "Year-Month-Day Hour:Minute:Second", where the Year is a 4-digit value, and all others are 2-digit values, of the functions defined in Calendar and Calendar.Formatting, including a leading zero, if needed. The separators between the values are a minus, another minus, a colon, and a single space between the Day and Hour. If Include_Time_Fraction is True, the integer part of Sub_Seconds*100 is suffixed to the string as a point followed by a 2-digit value. 82/2

Discussion: The Image provides a string in ISO 8601 format, the international standard time format. Alternative representations allowed in ISO 8601 are not supported here. 82.a/2

82.b/2 ISO 8601 allows 24:00:00 for midnight; and a seconds value of 60 for leap seconds. These are not allowed here (the routines mentioned above cannot produce those results).

82.c/2 **Ramification:** The fractional part is truncated, not rounded. It would be quite hard to define the result with proper rounding, as it can change all of the values of the image. Values can be rounded up by adding an appropriate constant (0.5 if Include_Time_Fraction is False, 0.005 otherwise) to the time before taking the image.

83/2
```
function Value (Date : String;
                Time_Zone  : Time_Zones.Time_Offset := 0) return Time;
```

84/2 {*AI95-00351-01*} Returns a Time value for the image given as Date, relative to the given time zone. Constraint_Error is raised if the string is not formatted as described for Image, or the function cannot interpret the given string as a Time value.

84.a/3 **Discussion:** {*AI05-0005-1*} The intent is that the implementation enforce the same range rules on the string as the appropriate function Time_Of, except for the hour, so "cannot interpret the given string as a Time value" happens when one of the values is out of the required range. For example, "2005-08-31 24:00:00" should raise Constraint_Error (the hour is out of range).

85/2
```
function Image (Elapsed_Time : Duration;
                Include_Time_Fraction : Boolean := False) return String;
```

86/5 {*AI95-00351-01*} {*AI12-0445-1*} Returns a string form of the Elapsed_Time. The format is "Hour:Minute:Second", where all values are 2-digit values, including a leading zero, if necessary. The separators between the values are colons. If Include_Time_Fraction is True, the integer part of Sub_Seconds*100 is suffixed to the string as a point followed by a 2-digit value. If Elapsed_Time < 0.0, the result is Image (**abs** Elapsed_Time, Include_Time_Fraction) prefixed with a minus sign. If **abs** Elapsed_Time represents 100 hours or more, the result is implementation-defined.

86.a/2 **Implementation defined:** The result of Calendar.Formatting.Image if its argument represents more than 100 hours.

86.b/2 **Implementation Note:** This cannot be implemented (directly) by calling Calendar.Formatting.Split, since it may be out of the range of Day_Duration, and thus the number of hours may be out of the range of Hour_Number.

86.c If a Duration value can represent more then 100 hours, the implementation will need to define a format for the return of Image.

87/2
```
function Value (Elapsed_Time : String) return Duration;
```

88/2 {*AI95-00351-01*} Returns a Duration value for the image given as Elapsed_Time. Constraint_Error is raised if the string is not formatted as described for Image, or the function cannot interpret the given string as a Duration value.

88.a/2 **Discussion:** The intent is that the implementation enforce the same range rules on the string as the appropriate function Time_Of, except for the hour, so "cannot interpret the given string as a Time value" happens when one of the values is out of the required range. For example, "10:23:60" should raise Constraint_Error (the seconds value is out of range).

Implementation Advice

89/2 {*AI95-00351-01*} An implementation should support leap seconds if the target system supports them. If leap seconds are not supported, Difference should return zero for Leap_Seconds, Split should return False for Leap_Second, and Time_Of should raise Time_Error if Leap_Second is True.

89.a/2 **Implementation Advice:** Leap seconds should be supported if the target system supports them. Otherwise, operations in Calendar.Formatting should return results consistent with no leap seconds.

89.b/2 **Discussion:** An implementation can always support leap seconds when the target system does not; indeed, this isn't particularly hard (all that is required is a table of when leap seconds were inserted). As such, leap second support isn't "impossible or impractical" in the sense of 1.1.3. However, for some purposes, it may be important to follow the target system's lack of leap second support (if the target is a GPS satellite, which does not use leap seconds, leap second support would be a handicap to work around). Thus, this Implementation Advice should be read as giving permission to not support leap seconds on target systems that don't support leap seconds. Implementers should use the needs of their customers to determine whether or not support leap seconds on such targets.

90/5 NOTE 1 {*AI95-00351-01*} {*AI12-0336-1*} {*AI12-0442-1*} The implementation-defined time zone of package Calendar can be the local time zone. Local_Time_Offset always returns the difference relative to the implementation-defined time zone

of package Calendar. If Local_Time_Offset does not raise Unknown_Zone_Error, UTC time can be safely calculated (within the accuracy of the underlying time-base).

Discussion: {*AI95-00351-01*} The time in the time zone known as Greenwich Mean Time (GMT) is generally very close to UTC time; for most purposes they can be treated the same. GMT is the time based on the rotation of the Earth; UTC is the time based on atomic clocks, with leap seconds periodically inserted to realign with GMT (because most human activities depend on the rotation of the Earth). At any point in time, there will be a sub-second difference between GMT and UTC. 90.a/2

NOTE 2 {*AI95-00351-01*} {*AI12-0336-1*} Calling Split on the results of subtracting Duration(Local_Time_Offset*60) from Clock provides the components (hours, minutes, and so on) of the UTC time. In the United States, for example, Local_Time_Offset will generally be negative. 91/5

Discussion: {*AI12-0336-1*} This is an illustration to help specify the value of Local_Time_Offset. A user should pass zero as the Time_Zone parameter of Split, rather than trying to make the above calculation. 91.a/5

Extensions to Ada 95

{*AI95-00351-01*} {*AI95-00428-01*} Packages Calendar.Time_Zones, Calendar.Arithmetic, and Calendar.Formatting are new. 91.b/2

Inconsistencies With Ada 2005

{*AI05-0238-1*} **Correction:** Defined that Split for Seconds raises Time_Error for a value of exactly 86400.0, rather than breaking some invariant or raising some other exception. Ada 2005 left this unspecified; a program that depended on what some implementation does might break, but such a program is not portable anyway. 91.c/3

Wording Changes from Ada 2005

{*AI05-0119-1*} **Correction:** Clarified the sign of UTC_Time_Offset. 91.d/3

Inconsistencies With Ada 2012

{*AI12-0336-1*} **Correction:** Changed the definition of Time_Offset to be compatible with most compilers. Also added Local_Time_Offset and Local_Image to better describe the intent. A program that expects the Ada 2012 definition of Time_Offset would get incorrect answers. However, most compilers tested use the revised definition, so the likelihood of a program breaking is quite low. Additionally, the new definitions could cause a use clause conflict; see the introduction of Annex A for more on this topic. 91.e/5

9.7 Select Statements

[There are four forms of the select_statement. One form provides a selective wait for one or more select_alternatives. Two provide timed and conditional entry calls. The fourth provides asynchronous transfer of control.] 1

Syntax

```
select_statement ::=
  selective_accept
 | timed_entry_call
 | conditional_entry_call
 | asynchronous_select
```
2

Examples

Example of a select statement: 3

```
select                                                    4
   accept Driver_Awake_Signal;
or
   delay 30.0*Seconds;
   Stop_The_Train;
end select;
```

Extensions to Ada 83

4.a Asynchronous_select is new.

9.7.1 Selective Accept

1 [This form of the select_statement allows a combination of waiting for, and selecting from, one or more alternatives. The selection may depend on conditions associated with each alternative of the selective_accept.]

Syntax

2
```
selective_accept ::=
  select
   [guard]
    select_alternative
{ or
   [guard]
    select_alternative }
[ else
   sequence_of_statements ]
  end select;
```

3 guard ::= **when** condition =>

4
```
select_alternative ::=
   accept_alternative
 | delay_alternative
 | terminate_alternative
```

5
```
accept_alternative ::=
  accept_statement [sequence_of_statements]
```

6
```
delay_alternative ::=
  delay_statement [sequence_of_statements]
```

7 terminate_alternative ::= **terminate**;

8 A selective_accept shall contain at least one accept_alternative. In addition, it can contain:

9 - a terminate_alternative (only one); or
10 - one or more delay_alternatives; or
11 - an *else part* (the reserved word **else** followed by a sequence_of_statements).

12 These three possibilities are mutually exclusive.

Legality Rules

13 If a selective_accept contains more than one delay_alternative, then all shall be delay_relative_-statements, or all shall be delay_until_statements for the same time type.

13.a **Reason:** This simplifies the implementation and the description of the semantics.

Dynamic Semantics

14 A select_alternative is said to be *open* if it is not immediately preceded by a guard, or if the condition of its guard evaluates to True. It is said to be *closed* otherwise.

For the execution of a selective_accept, any guard conditions are evaluated; open alternatives are thus determined. For an open delay_alternative, the *delay*_expression is also evaluated. Similarly, for an open accept_alternative for an entry of a family, the entry_index is also evaluated. These evaluations are performed in an arbitrary order, except that a *delay*_expression or entry_index is not evaluated until after evaluating the corresponding condition, if any. Selection and execution of one open alternative, or of the else part, then completes the execution of the selective_accept; the rules for this selection are described below. 15

Open accept_alternatives are first considered. Selection of one such alternative takes place immediately if the corresponding entry already has queued calls. If several alternatives can thus be selected, one of them is selected according to the entry queuing policy in effect (see 9.5.3 and D.4). When such an alternative is selected, the selected call is removed from its entry queue and the handled_sequence_of_-statements (if any) of the corresponding accept_statement is executed; after the rendezvous completes any subsequent sequence_of_statements of the alternative is executed. If no selection is immediately possible (in the above sense) and there is no else part, the task blocks until an open alternative can be selected. 16

Selection of the other forms of alternative or of an else part is performed as follows: 17

- An open delay_alternative is selected when its expiration time is reached if no accept_-alternative or other delay_alternative can be selected prior to the expiration time. If several delay_alternatives have this same expiration time, one of them is selected according to the queuing policy in effect (see D.4); the default queuing policy chooses arbitrarily among the delay_alternatives whose expiration time has passed. 18
- The else part is selected and its sequence_of_statements is executed if no accept_alternative can immediately be selected; in particular, if all alternatives are closed. 19
- {*AI05-0299-1*} {*AI12-0449-1*} An open terminate_alternative is selected if the conditions stated at the end of 9.3 are satisfied. 20/5

  Ramification: In the absence of a requeue_statement, the conditions stated are such that a terminate_alternative cannot be selected while there is a queued entry call for any entry of the task. In the presence of requeues from a task to one of its subtasks, it is possible that when a terminate_alternative of the subtask is selected, requeued calls (for closed entries only) might still be queued on some entry of the subtask. Tasking_Error will be propagated to such callers, as is usual when a task completes while queued callers remain. 20.a

The exception Program_Error is raised if all alternatives are closed and there is no else part. 21

NOTE {*AI12-0447-1*} A selective_accept can have several open delay_alternatives. A selective_accept can have several open accept_alternatives for the same entry. 22/5

Examples

Example of a task body with a selective accept: 23

```
task body Server is                                                    24
   Current_Work_Item : Work_Item;
begin
   loop
      select
         accept Next_Work_Item(WI : in Work_Item) do
            Current_Work_Item := WI;
         end;
         Process_Work_Item(Current_Work_Item);
      or
         accept Shut_Down;
         exit;          -- Premature shut down requested
      or
         terminate;     -- Normal shutdown at end of scope
      end select;
   end loop;
end Server;
```

Wording Changes from Ada 83

24.a The name of selective_wait was changed to selective_accept to better describe what is being waited for. We kept select_alternative as is, because selective_accept_alternative was too easily confused with accept_alternative.

9.7.2 Timed Entry Calls

1/5 {*AI95-00345-01*} {*AI12-0439-1*} [A timed_entry_call issues an entry call that is cancelled if the call (or a requeue-with-abort of the call) is not selected before the expiration time is reached. A procedure call may appear rather than an entry call for cases where the procedure can be implemented by an entry.]

Syntax

2
```
timed_entry_call ::=
 select
  entry_call_alternative
 or
  delay_alternative
 end select;
```

3/2 {*AI95-00345-01*} entry_call_alternative ::=
 procedure_or_entry_call [sequence_of_statements]

3.1/2 {*AI95-00345-01*} procedure_or_entry_call ::=
 procedure_call_statement | entry_call_statement

Legality Rules

3.2/2 {*AI95-00345-01*} If a procedure_call_statement is used for a procedure_or_entry_call, the *procedure_*name or *procedure_*prefix of the procedure_call_statement shall statically denote an entry renamed as a procedure or (a view of) a primitive subprogram of a limited interface whose first parameter is a controlling parameter (see 3.9.2).

3.a/2 **Reason:** This would be a confusing way to call a procedure, so we only allow it when it is possible that the procedure is actually an entry. We could have allowed formal subprograms here, but we didn't because we'd have to allow all formal subprograms, and it would increase the difficulty of generic code sharing.

3.b/2 We say “statically denotes” because an access-to-subprogram cannot be primitive, and we don't have anything like access-to-entry. So only names of entries or procedures are possible.

Dynamic Semantics

4/2 {*AI95-00345-01*} For the execution of a timed_entry_call, the *entry_*name, *procedure_*name, or *procedure_*prefix, and any actual parameters are evaluated, as for a simple entry call (see 9.5.3) or procedure call (see 6.4). The expiration time (see 9.6) for the call is determined by evaluating the *delay_*expression of the delay_alternative. If the call is an entry call or a call on a procedure implemented by an entry, the entry call is then issued. Otherwise, the call proceeds as described in 6.4 for a procedure call, followed by the sequence_of_statements of the entry_call_alternative; the sequence_of_statements of the delay_alternative is ignored.

5 If the call is queued (including due to a requeue-with-abort), and not selected before the expiration time is reached, an attempt to cancel the call is made. If the call completes due to the cancellation, the optional sequence_of_statements of the delay_alternative is executed; if the entry call completes normally, the optional sequence_of_statements of the entry_call_alternative is executed.

5.a/2 *This paragraph was deleted.*{*AI95-00345-01*}

Examples

Example of a timed entry call: 6

```
select                                                                    7
   Controller.Request(Medium)(Some_Item);
or
   delay 45.0;
   -- controller too busy, try something else
end select;
```

Wording Changes from Ada 83

{*AI05-0299-1*} This subclause comes before the one for Conditional Entry Calls, so we can define conditional entry calls in terms of timed entry calls. 7.a/3

Incompatibilities With Ada 95

{*AI95-00345-01*} {*AI05-0005-1*} A procedure call can be used as the entry_call_alternative in a timed or conditional entry call, if the procedure might actually be an entry. Since the fact that something is an entry could be used in resolving these calls in Ada 95, it is possible for timed or conditional entry calls that resolved in Ada 95 to be ambiguous in Ada 2005. That could happen if both an entry and procedure with the same name and profile exist, which should be rare. 7.b/3

9.7.3 Conditional Entry Calls

{*AI95-00345-01*} {*AI12-0439-1*} [A conditional_entry_call issues an entry call that is then cancelled if it is not selected immediately (or if a requeue-with-abort of the call is not selected immediately). A procedure call may appear rather than an entry call for cases where the procedure can be implemented by an entry.] 1/5

To be honest: In the case of an entry call on a protected object, it is OK if the entry is closed at the start of the corresponding protected action, so long as it opens and the call is selected before the end of that protected action (due to changes in the Count attribute). 1.a

Syntax

```
conditional_entry_call ::=                                                2
 select
  entry_call_alternative
 else
  sequence_of_statements
 end select;
```

Dynamic Semantics

The execution of a conditional_entry_call is defined to be equivalent to the execution of a timed_entry_call with a delay_alternative specifying an immediate expiration time and the same sequence_of_statements as given after the reserved word **else**. 3

NOTE {*AI12-0440-1*} A conditional_entry_call can briefly increase the Count attribute of the entry, even if the conditional call is not selected. 4/5

Examples

5 *Example of a conditional entry call:*

6/5
```
{AI12-0178-1} procedure Spin(R : in out Resource) is  -- see 9.4
begin
   loop
      select
         R.Seize;
         return;
      else
         null;  -- busy waiting
      end select;
   end loop;
end;
```

Wording Changes from Ada 83

6.a/3 {*AI05-0299-1*} This subclause comes after the one for Timed Entry Calls, so we can define conditional entry calls in terms of timed entry calls. We do that so that an "expiration time" is defined for both, thereby simplifying the definition of what happens on a requeue-with-abort.

9.7.4 Asynchronous Transfer of Control

1 [An asynchronous select_statement provides asynchronous transfer of control upon completion of an entry call or the expiration of a delay.]

Syntax

2
```
asynchronous_select ::=
 select
  triggering_alternative
 then abort
  abortable_part
 end select;
```

3 triggering_alternative ::= triggering_statement [sequence_of_statements]

4/2 {*AI95-00345-01*} triggering_statement ::= procedure_or_entry_call | delay_statement

5 abortable_part ::= sequence_of_statements

Dynamic Semantics

6/2 {*AI95-00345-01*} For the execution of an asynchronous_select whose triggering_statement is a procedure_or_entry_call, the *entry*_name, *procedure*_name, or *procedure*_prefix, and actual parameters are evaluated as for a simple entry call (see 9.5.3) or procedure call (see 6.4). If the call is an entry call or a call on a procedure implemented by an entry, the entry call is issued. If the entry call is queued (or requeued-with-abort), then the abortable_part is executed. [If the entry call is selected immediately, and never requeued-with-abort, then the abortable_part is never started.] If the call is on a procedure that is not implemented by an entry, the call proceeds as described in 6.4, followed by the sequence_of_-statements of the triggering_alternative[; the abortable_part is never started].

7 For the execution of an asynchronous_select whose triggering_statement is a delay_statement, the *delay*_expression is evaluated and the expiration time is determined, as for a normal delay_statement. If the expiration time has not already passed, the abortable_part is executed.

8 If the abortable_part completes and is left prior to completion of the triggering_statement, an attempt to cancel the triggering_statement is made. If the attempt to cancel succeeds (see 9.5.3 and 9.6), the asynchronous_select is complete.

If the triggering_statement completes other than due to cancellation, the abortable_part is aborted (if started but not yet completed — see 9.8). If the triggering_statement completes normally, the optional sequence_of_statements of the triggering_alternative is executed after the abortable_part is left. 9

Discussion: We currently don't specify when the by-copy [**in**] **out** parameters are assigned back into the actuals. We considered requiring that to happen after the abortable_part is left. However, that doesn't seem useful enough to justify possibly overspecifying the implementation approach, since some of the parameters are passed by reference anyway. 9.a

In an earlier description, we required that the sequence_of_statements of the triggering_alternative execute after aborting the abortable_part, but before waiting for it to complete and finalize, to provide more rapid response to the triggering event in case the finalization was unbounded. However, various reviewers felt that this created unnecessary complexity in the description, and a potential for undesirable concurrency (and nondeterminism) within a single task. We have now reverted to simpler, more deterministic semantics, but anticipate that further discussion of this issue might be appropriate during subsequent reviews. One possibility is to leave this area implementation defined, so as to encourage experimentation. The user would then have to assume the worst about what kinds of actions are appropriate for the sequence_of_statements of the triggering_alternative to achieve portability. 9.b

Examples

Example of a main command loop for a command interpreter: 10

```
loop                                                                    11
   select
      Terminal.Wait_For_Interrupt;
      Put_Line("Interrupted");
   then abort
      -- This will be abandoned upon terminal interrupt
      Put_Line("-> ");
      Get_Line(Command, Last);
      Process_Command(Command(1..Last));
   end select;
end loop;
```

Example of a time-limited calculation: 12

```
{AI12-0442-1} select                                                    13/5
   delay 5.0;
   Put_Line("Calculation does not converge");
then abort
   -- This calculation is expected to finish in 5.0 seconds;
   --  if not, it is assumed to diverge.
   Horribly_Complicated_Recursive_Function(X, Y);
end select;
```

{*AI12-0098-1*} {*AI12-0418-1*} Note that these examples presume that there are abort completion points (see 9.8) within the execution of the abortable_part. 14/5

Extensions to Ada 83

Asynchronous_select is new. 14.a

Extensions to Ada 95

{*AI95-00345-01*} A procedure can be used as the triggering_statement of an asynchronous_select, if the procedure might actually be an entry. 14.b/2

9.8 Abort of a Task - Abort of a Sequence of Statements

[An abort_statement causes one or more tasks to become abnormal, thus preventing any further interaction with such tasks. The completion of the triggering_statement of an asynchronous_select causes a sequence_of_statements to be aborted.] 1

Syntax

abort_statement ::= **abort** *task*_name {, *task*_name}; 2

Name Resolution Rules

3/5 {*AI12-0444-1*} Each *task_*name is expected to be of any task type[; each can be of a different task type.]

Dynamic Semantics

4 For the execution of an abort_statement, the given *task_*names are evaluated in an arbitrary order. Each named task is then *aborted*, which consists of making the task *abnormal* and aborting the execution of the corresponding task_body, unless it is already completed.

4.a/2 **Ramification:** {*AI95-00114-01*} Note that aborting those tasks is not defined to be an abort-deferred operation. Therefore, if one of the named tasks is the task executing the abort_statement, or if the task executing the abort_statement depends on one of the named tasks, then it is possible for the execution of the abort_statement to be aborted, thus leaving some of the tasks unaborted. This allows the implementation to use either a sequence of calls to an "abort task" run-time system primitive, or a single call to an "abort list of tasks" run-time system primitive.

5 When the execution of a construct is *aborted* (including that of a task_body or of a sequence_of_-statements), the execution of every construct included within the aborted execution is also aborted, except for executions included within the execution of an *abort-deferred* operation; the execution of an abort-deferred operation continues to completion without being affected by the abort; the following are the abort-deferred operations:

6 • a protected action;

7 • waiting for an entry call to complete (after having initiated the attempt to cancel it — see below);

8 • waiting for the termination of dependent tasks;

9 • the execution of an Initialize procedure as the last step of the default initialization of a controlled object;

10 • the execution of a Finalize procedure as part of the finalization of a controlled object;

11 • an assignment operation to an object with a controlled part.

12 [The last three of these are discussed further in 7.6.]

12.a **Reason:** Deferring abort during Initialize and finalization allows, for example, the result of an allocator performed in an Initialize operation to be assigned into an access object without being interrupted in the middle, which would cause storage leaks. For an object with several controlled parts, each individual Initialize is abort-deferred. Note that there is generally no semantic difference between making each Finalize abort-deferred, versus making a group of them abort-deferred, because if the task gets aborted, the first thing it will do is complete any remaining finalizations. Individual objects are finalized prior to an assignment operation (if nonlimited controlled) and as part of Unchecked_Deallocation.

12.b **Ramification:** Abort is deferred during the entire assignment operation to an object with a controlled part, even if only some subcomponents are controlled. Note that this says "assignment operation", not "assignment_statement". Explicit calls to Initialize, Finalize, or Adjust are not abort-deferred.

13 When a master is aborted, all tasks that depend on that master are aborted.

14 The order in which tasks become abnormal as the result of an abort_statement or the abort of a sequence_of_statements is not specified by the language.

15 If the execution of an entry call is aborted, an immediate attempt is made to cancel the entry call (see 9.5.3). If the execution of a construct is aborted at a time when the execution is blocked, other than for an entry call, at a point that is outside the execution of an abort-deferred operation, then the execution of the construct completes immediately. For an abort due to an abort_statement, these immediate effects occur before the execution of the abort_statement completes. Other than for these immediate cases, the execution of a construct that is aborted does not necessarily complete before the abort_statement completes. However, the execution of the aborted construct completes no later than its next *abort completion point* (if any) that occurs outside of an abort-deferred operation; the following are abort completion points for an execution:

- the point where the execution initiates the activation of another task; 16
- the end of the activation of a task; 17
- {*AI12-0119-1*} a point within a parallel construct where a new logical thread of control is created; 17.1/5
- {*AI12-0119-1*} the end of a parallel construct; 17.2/5
- the start or end of the execution of an entry call, accept_statement, delay_statement, or abort_statement; 18

  Ramification: Although the abort completion point doesn't occur until the end of the entry call or delay_statement, these operations might be cut short because an abort attempts to cancel them. 18.a
- the start of the execution of a select_statement, or of the sequence_of_statements of an exception_handler. 19

  Reason: The start of an exception_handler is considered an abort completion point simply because it is easy for an implementation to check at such points. 19.a

  Implementation Note: Implementations may of course check for abort more often than at each abort completion point; ideally, a fully preemptive implementation of abort will be provided. If preemptive abort is not supported in a given environment, then supporting the checking for abort as part of subprogram calls and loop iterations might be a useful option. 19.b

Bounded (Run-Time) Errors

{*AI05-0264-1*} {*AI12-0445-1*} An attempt to execute an asynchronous_select as part of the execution of an abort-deferred operation is a bounded error. Similarly, an attempt to create a task that depends on a master that is included entirely within the execution of an abort-deferred operation is a bounded error. In both cases, Program_Error is raised if the error is detected by the implementation; otherwise, the operations proceed as they would outside an abort-deferred operation, except that an abort of the abortable_part or the created task does not necessarily have an effect. 20/5

Reason: An asynchronous_select relies on an abort of the abortable_part to effect the asynchronous transfer of control. For an asynchronous_select within an abort-deferred operation, the abort might have no effect. 20.a

Creating a task dependent on a master included within an abort-deferred operation is considered an error, because such tasks could be aborted while the abort-deferred operation was still progressing, undermining the purpose of abort-deferral. Alternatively, we could say that such tasks are abort-deferred for their entire execution, but that seems too easy to abuse. Note that task creation is already a bounded error in protected actions, so this additional rule only applies to local task creation as part of Initialize, Finalize, or Adjust. 20.b

Erroneous Execution

If an assignment operation completes prematurely due to an abort, the assignment is said to be *disrupted*; the target of the assignment or its parts can become abnormal, and certain subsequent uses of the object can be erroneous, as explained in 13.9.1. 21

NOTE 1 {*AI12-0442-1*} An abort_statement is best used only in situations requiring unconditional termination. 22/5

NOTE 2 A task is allowed to abort any task it can name, including itself. 23

NOTE 3 {*AI12-0449-1*} Additional requirements associated with abort are given in D.6. 24/5

Wording Changes from Ada 83

{*AI05-0299-1*} This subclause has been rewritten to accommodate the concept of aborting the execution of a construct, rather than just of a task. 24.a/3

Wording Changes from Ada 2012

{*AI12-0119-1*} Added points within parallel constructs to the list of abort completion points. 24.b/5

9.9 Task and Entry Attributes

Dynamic Semantics

1 For a prefix T that is of a task type [(after any implicit dereference)], the following attributes are defined:

2 T'Callable Yields the value True when the task denoted by T is *callable*, and False otherwise; a task is callable unless it is completed or abnormal. The value of this attribute is of the predefined type Boolean.

3 T'Terminated
Yields the value True if the task denoted by T is terminated, and False otherwise. The value of this attribute is of the predefined type Boolean.

4 For a prefix E that denotes an entry of a task or protected unit, the following attribute is defined. This attribute is only allowed within the body of the task or protected unit, but excluding, in the case of an entry of a task unit, within any program unit that is, itself, inner to the body of the task unit.

5 E'Count Yields the number of calls presently queued on the entry E of the current instance of the unit. The value of this attribute is of the type *universal_integer*.

6 NOTE 1 For the Count attribute, the entry can be either a single entry or an entry of a family. The name of the entry or entry family can be either a direct_name or an expanded name.

7/5 NOTE 2 {*AI12-0442-1*} Within task units, by interrogating the attribute E'Count an algorithm can allow for the increase of the value of this attribute for incoming entry calls, and its decrease, for example with timed_entry_calls. A conditional_entry_call can also briefly increase this value, even if the conditional call is not accepted.

8/5 NOTE 3 {*AI12-0442-1*} Within protected units, by interrogating the attribute E'Count in the entry_barrier for the entry E an algorithm can allow for the evaluation of the condition of the barrier both before and after queuing a given caller.

9.10 Shared Variables

Static Semantics

1/5 {*AI05-0009-1*} {*AI05-0201-1*} {*AI05-0229-1*} {*AI05-0295-1*} {*AI12-0119-1*} {*AI12-0363-1*} If two different objects, including nonoverlapping parts of the same object, are *independently addressable*, they can be manipulated concurrently by two different logical threads of control without synchronization, unless both are subcomponents of the same full access object, and either is nonatomic (see C.6). Any two nonoverlapping objects are independently addressable if either object is specified as independently addressable (see C.6). Otherwise, two nonoverlapping objects are independently addressable except when they are both parts of a composite object for which a nonconfirming value is specified for any of the following representation aspects: (record) Layout, Component_Size, Pack, Atomic, or Convention; in this case it is unspecified whether the parts are independently addressable.

1.a/3 *This paragraph was deleted.*

1.b/3 **Implementation Note:** {*AI05-0229-1*} Independent addressability is the only high level semantic effect of aspect Pack. If two objects are independently addressable, the implementation should allocate them in such a way that each can be written by the hardware without writing the other. For example, unless the user asks for it, it is generally not feasible to choose a bit-packed representation on a machine without an atomic bit field insertion instruction, because there might be tasks that update neighboring subcomponents concurrently, and locking operations on all subcomponents is generally not a good idea.

1.c/3 {*AI05-0229-1*} Even if Pack or one of the other above-mentioned aspects is specified, subcomponents should still be updated independently if the hardware efficiently supports it.

1.d/4 **Ramification:** {*AI05-0009-1*} {*AI05-0201-1*} {*AI12-0001-1*} An atomic object (including atomic components) is always independently addressable from any other nonoverlapping object. Aspect_specifications and representation items cannot change that fact. Note, however, that the components of an atomic object are not necessarily atomic.

Dynamic Semantics

{*AI12-0119-1*} [Separate logical threads of control normally proceed independently and concurrently with one another. However, task interactions can be used to synchronize the actions of two or more logical threads of control to allow, for example, meaningful communication by the direct updating and reading of variables shared between them.] The actions of two different logical threads of control are synchronized in this sense when an action of one *signals* an action of the other; an action A1 is defined to signal an action A2 under the following circumstances: 2/5

- If A1 and A2 are part of the execution of the same task, and the language rules require A1 to be performed before A2; 3
- If A1 is the action of an activator that initiates the activation of a task, and A2 is part of the execution of the task that is activated; 4
- If A1 is part of the activation of a task, and A2 is the action of waiting for completion of the activation; 5
- If A1 is part of the execution of a task, and A2 is the action of waiting for the termination of the task; 6
- {*8652/0031*} {*AI95-00118-01*} {*AI05-0072-1*} If A1 is the termination of a task T, and A2 is either an evaluation of the expression T'Terminated that results in True, or a call to Ada.Task_Identification.Is_Terminated with an actual parameter that identifies T and a result of True (see C.7.1); 6.1/3
- {*AI05-0262-1*} If A1 is the action of issuing an entry call, and A2 is part of the corresponding execution of the appropriate entry_body or accept_statement; 7/3

  Ramification: Evaluating the entry_index of an accept_statement is not synchronized with a corresponding entry call, nor is evaluating the entry barrier of an entry_body. 7.a
- If A1 is part of the execution of an accept_statement or entry_body, and A2 is the action of returning from the corresponding entry call; 8
- If A1 is part of the execution of a protected procedure body or entry_body for a given protected object, and A2 is part of a later execution of an entry_body for the same protected object; 9

  Reason: The underlying principle here is that for one action to “signal” a second, the second action has to follow a potentially blocking operation, whose blocking is dependent on the first action in some way. Protected procedures are not potentially blocking, so they can only be "signalers", they cannot be signaled. 9.a

  Ramification: Protected subprogram calls are not defined to signal one another, which means that such calls alone cannot be used to synchronize access to shared data outside of a protected object. 9.b

  Reason: The point of this distinction is so that on multiprocessors with inconsistent caches, the caches only need to be refreshed at the beginning of an entry body, and forced out at the end of an entry body or protected procedure that leaves an entry open. Protected function calls, and protected subprogram calls for entryless protected objects do not require full cache consistency. Entryless protected objects are intended to be treated roughly like atomic objects — each operation is indivisible with respect to other operations (unless both are reads), but such operations cannot be used to synchronize access to other nonvolatile shared variables. 9.c
- If A1 signals some action that in turn signals A2. 10

{*AI12-0298-1*} Action A1 is defined to *potentially signal* action A2 if A1 signals A2, if action A1 and A2 occur as part of the execution of the same logical thread of control, and the language rules permit action A1 to precede action A2, or if action A1 potentially signals some action that in turn potentially signals A2. 10.1/5

{*AI12-0267-1*} Two actions are defined to be *sequential* if one of the following is true: 11/5

- One action signals the other; 12
- {*AI12-0119-1*} Both actions occur as part of the execution of the same logical thread of control; 13/5

13.a/5 **Reason:** {*AI12-0005-1*} {*AI12-0119-1*} Any two actions of the same logical thread of control are sequential, even if one does not signal the other because they can be executed in an "arbitrary" (but necessarily equivalent to some "sequential") order.

14/5 - {*AI12-0292-1*} Both actions occur as part of protected actions on the same protected object, and at least one of the actions is part of a call on an exclusive protected operation of the protected object.

14.a/5 **Reason:** {*AI12-0292-1*} Because actions within protected actions do not always imply signaling, we have to mention them here explicitly to make sure that actions occurring within different protected actions of the same protected object are sequential with respect to one another (unless both are part of calls on nonexclusive protected functions).

14.b/5 **Ramification:** {*AI12-0292-1*} It doesn't matter whether or not the variable being assigned is actually a subcomponent of the protected object; globals can be safely updated from within the bodies of protected procedures, protected entries, or exclusive protected functions.

15/3 {*AI05-0229-1*} Aspect Atomic or aspect Atomic_Components may also be specified to ensure that certain reads and updates are sequential — see C.6.

15.a/5 **Ramification:** {*AI12-0119-1*} If two actions are "sequential" it is known that their executions don't overlap in time, but it is not necessarily specified which occurs first. For example, all actions of a single logical thread of control are sequential, even though the exact order of execution is not fully specified for all constructs.

15.b **Discussion:** Note that if two assignments to the same variable are sequential, but neither signals the other, then the program is not erroneous, but it is not specified which assignment ultimately prevails. Such a situation usually corresponds to a programming mistake, but in some (rare) cases, the order makes no difference, and for this reason this situation is not considered erroneous nor even a bounded error. In Ada 83, this was considered an "incorrect order dependence" if the "effect" of the program was affected, but "effect" was never fully defined. In Ada 95, this situation represents a potential nonportability, and a friendly compiler might want to warn the programmer about the situation, but it is not considered an error. An example where this would come up would be in gathering statistics as part of referencing some information, where the assignments associated with statistics gathering don't need to be ordered since they are just accumulating aggregate counts, sums, products, etc.

16/5 {*AI12-0267-1*} Two actions that are not sequential are defined to be *concurrent* actions.

17/5 {*AI12-0267-1*} {*AI12-0298-1*} Two actions are defined to *conflict* if one action assigns to an object, and the other action reads or assigns to a part of the same object (or of a neighboring object if the two are not independently addressable). The action comprising a call on a subprogram or an entry is defined to *potentially conflict* with another action if the Global aspect (or Global'Class aspect in the case of a dispatching call) of the called subprogram or entry is such that a conflicting action would be possible during the execution of the call. Similarly, two calls are considered to potentially conflict if they each have Global (or Global'Class in the case of a dispatching call) aspects such that conflicting actions would be possible during the execution of the calls. Finally, two actions that conflict are also considered to potentially conflict.

18/5 {*AI12-0267-1*} A *synchronized* object is an object of a task or protected type, an atomic object (see C.6), a suspension object (see D.10), or a synchronous barrier (see D.10.1). [Operations on such objects are necessarily sequential with respect to one another, and hence are never considered to conflict.]

Erroneous Execution

19/5 {*AI12-0267-1*} The execution of two concurrent actions is erroneous if the actions make conflicting uses of a shared variable (or neighboring variables that are not independently addressable).

Wording Changes from Ada 95

19.a/2 {*8652/0031*} {*AI95-00118-01*} **Corrigendum:** Clarified that a task T2 can rely on values of variables that are updated by another task T1, if task T2 first verifies that T1'Terminated is True.

Wording Changes from Ada 2005

19.b/3 {*AI05-0009-1*} {*AI05-0201-1*} **Correction:** Revised the definition of independent addressability to exclude conforming representation clauses and to require that atomic and independent objects always have independent addressability. This should not change behavior that the user sees for any Ada program, so it is not an inconsistency.

{*AI05-0072-1*} **Correction:** Corrected the wording of AI95-00118-01 to actually say what was intended (as described above). 19.c/3

Wording Changes from Ada 2012

{*AI12-0119-1*} Added wording to support interaction of parallel constructs with tasks by changing various wording to talk about logical threads of control rather than purely about tasks. 19.d/5

{*AI12-0267-1*} {*AI12-0298-1*} Added wording to define potentially signalling actions and conflicting actions; these are used to define conflict policies. 19.e/5

9.10.1 Conflict Check Policies

{*AI12-0267-1*} This subclause determines what checks are performed relating to possible concurrent conflicting actions (see 9.10). 1/5

Syntax

{*AI12-0267-1*} {*AI12-0298-1*} The form of a pragma Conflict_Check_Policy is as follows: 2/5

pragma Conflict_Check_Policy (*policy_*identifier[, *policy_*identifier]); 3/5

{*AI12-0267-1*} A pragma Conflict_Check_Policy is allowed only immediately within a declarative_part, a package_specification, or as a configuration pragma. 4/5

Legality Rules

{*AI12-0267-1*} Each *policy_*identifier shall be one of No_Parallel_Conflict_Checks, Known_Parallel_Conflict_Checks, All_Parallel_Conflict_Checks, No_Tasking_Conflict_Checks, Known_Tasking_Conflict_Checks, All_Tasking_Conflict_Checks, No_Conflict_Checks, Known_Conflict_Checks, All_Conflict_Checks, or an implementation-defined conflict check policy. If two *policy_*identifiers are given, one shall include the word Parallel and one shall include the word Tasking. If only one *policy_*identifier is given, it shall not include the word Parallel or Tasking. 5/5

Implementation defined: Implementation-defined conflict check policies. 5.a.1/5

{*AI12-0267-1*} A pragma Conflict_Check_Policy given in a declarative_part or immediately within a package_specification applies from the place of the pragma to the end of the innermost enclosing declarative region. The region for a pragma Conflict_Check_Policy given as a configuration pragma is the declarative region for the entire compilation unit (or units) to which it applies. 6/5

{*AI12-0267-1*} If a pragma Conflict_Check_Policy applies to a generic_instantiation, then the pragma Conflict_Check_Policy applies to the entire instance. 7/5

{*AI12-0267-1*} {*AI12-0294-1*} {*AI12-0298-1*} If multiple Conflict_Check_Policy pragmas apply to a given construct, the conflict check policy is determined by the one in the innermost enclosing region. If no Conflict_Check_Policy pragma applies to a construct, the policy is (All_Parallel_Conflict_Checks, No_Tasking_Conflict_Checks) (see below). 8/5

To be honest: {*AI12-0294-1*} The region mentioned in this rule is the region to which the policy pragma applies, and not the declarative region in which the policy pragma appears. This distinction matters when there are multiple policy pragmas in a single declarative region. 8.a/5

{*AI12-0267-1*} {*AI12-0298-1*} Certain potentially conflicting actions are disallowed according to which conflict check policies apply at the place where the action or actions occur, as follows: 9/5

No_Parallel_Conflict_Checks 10/5

This policy imposes no restrictions on concurrent actions arising from parallel constructs.

No_Tasking_Conflict_Checks 11/5

This policy imposes no restrictions on concurrent actions arising from tasking constructs.

12/5 Known_Parallel_Conflict_Checks

If this policy applies to two concurrent actions appearing within parallel constructs, they are disallowed if they are known to denote the same object (see 6.4.1) with uses that conflict. For the purposes of this check, any parallel loop may be presumed to involve multiple concurrent iterations. Also, for the purposes of deciding whether two actions are concurrent, it is enough for the logical threads of control in which they occur to be concurrent at any point in their execution, unless all of the following are true:

13/5 • the shared object is volatile;

14/5 • the two logical threads of control are both known to also refer to a shared synchronized object; and

15/5 • each thread whose potentially conflicting action updates the shared volatile object, also updates this shared synchronized object.

15.a/5 **Reason:** To properly synchronize two actions to prevent concurrency, a thread that does an update of a volatile object must update a synchronized object afterwards to indicate it has completed its update, and the other thread needs to test the value of the synchronized object before it reads the updated volatile object. In a parallel construct, "signaling" cannot be used to prevent concurrency, since that generally requires some blocking, so testing the value of the synchronized object would probably need to use a busy-wait loop.

16/5 Known_Tasking_Conflict_Checks

If this policy applies to two concurrent actions appearing within the same compilation unit, at least one of which appears within a task body but not within a parallel construct, they are disallowed if they are known to denote the same object (see 6.4.1) with uses that conflict, and neither potentially signals the other (see 9.10). For the purposes of this check, any named task type may be presumed to have multiple instances. Also, for the purposes of deciding whether two actions are concurrent, it is enough for the tasks in which they occur to be concurrent at any point in their execution, unless all of the following are true:

17/5 • the shared object is volatile;

18/5 • the two tasks are both known to also refer to a shared synchronized object; and

19/5 • each task whose potentially conflicting action updates the shared volatile object, also updates this shared synchronized object.

19.a/5 **Reason:** To properly synchronize two actions to prevent concurrency, a task that does an update of a *non*volatile object must use signaling via a synchronized object to indicate it has completed its update, and the other task needs to be signaled by this action on the synchronized object before it reads the updated nonvolatile object. In other words, to synchronize communication via a nonvolatile object, signaling must be used. To synchronize communication via a volatile object, an update of a shared synchronized object followed by a read of the synchronized object in the other task can be sufficient.

20/5 All_Parallel_Conflict_Checks

This policy includes the restrictions imposed by the Known_Parallel_Conflict_Checks policy, and in addition disallows a parallel construct from reading or updating a variable that is global to the construct, unless it is a synchronized object, or unless the construct is a parallel loop, and the global variable is a part of a component of an array denoted by an indexed component with at least one index expression that statically denotes the loop parameter of the loop_parameter_specification or the chunk parameter of the parallel loop.

21/5 All_Tasking_Conflict_Checks

This policy includes the restrictions imposed by the Known_Tasking_Conflict_Checks policy, and in addition disallows a task body from reading or updating a variable that is global to the task body, unless it is a synchronized object.

22/5 No_Conflict_Checks, Known_Conflict_Checks, All_Conflict_Checks

These are shorthands for (No_Parallel_Conflict_Checks, No_Tasking_Conflict_Checks), (Known_Parallel_Conflict_Checks, Known_Tasking_Conflict_Checks), and (All_Parallel_Conflict_Checks, All_Tasking_Conflict_Checks), respectively.

Static Semantics

{*AI12-0344-1*} For a subprogram, the following language-defined representation aspect may be specified: 23/5

Parallel_Calls 24/5

The Parallel_Calls aspect is of type Boolean. The specified value shall be static. The Parallel_Calls aspect of an inherited primitive subprogram is True if Parallel_Calls is True either for the corresponding subprogram of the progenitor type or for any other inherited subprogram that it overrides. If not specified or inherited as True, the Parallel_Calls aspect of a subprogram is False.

Specifying the Parallel_Calls aspect to be True for a subprogram indicates that the subprogram can be safely called in parallel. Conflict checks (if required by the Conflict_Check_Policy in effect) are made on the subprogram assuming that multiple concurrent calls exist.[Such checks can then be omitted on a call of the subprogram in a parallel iteration context.] 25/5

Aspect Description for Parallel_Calls: Specifies whether a given subprogram is expected to be called in parallel. 25.a/5

Implementation Permissions

{*AI12-0267-1*} {*AI12-0298-1*} When the conflict check policy Known_Parallel_Conflict_Checks or All_Parallel_Conflict_Checks applies, the implementation may disallow two concurrent actions appearing within parallel constructs if the implementation can prove they will at run-time denote the same object with uses that conflict. Similarly, when the conflict check policy Known_Tasking_Conflict_Checks or All_Tasking_Conflict_Checks applies, the implementation may disallow two concurrent actions, at least one of which appears within a task body but not within a parallel construct, if the implementation can prove they will at run-time denote the same object with uses that conflict. 26/5

Ramification: This permission allows additional enforcement in instance bodies (where Legality Rules aren't usually enforced), in subunits and their parents, and across compilation units, if the implementation wishes. 26.a/5

Extensions to Ada 2012

{*AI12-0267-1*} Conflict check policies (and the associated checks) are new. 26.b/5

{*AI12-0344-1*} Aspect Parallel_Calls is new. 26.c/5

9.11 Example of Tasking and Synchronization

Examples

{*AI12-0440-1*} The following example defines a buffer protected object to smooth variations between the speed of output of a producing task and the speed of input of some consuming task. For instance, the producing task can have the following structure: 1/5

```
task Producer;                                                    2
{AI95-00433-01} task body Producer is                             3/2
   Person : Person_Name; -- see 3.10.1
begin
   loop
      ... -- simulate arrival of the next customer
      Buffer.Append_Wait(Person);
      exit when Person = null;
   end loop;
end Producer;
```

{*AI12-0440-1*} and the consuming task can have the following structure: 4/5

```
task Consumer;                                                    5
```

6/2
```
{AI95-00433-01} task body Consumer is
   Person : Person_Name;
begin
   loop
      Buffer.Remove_First_Wait(Person);
      exit when Person = null;
      ... -- simulate serving a customer
   end loop;
end Consumer;
```

7/2 {*AI95-00433-01*} The buffer object contains an internal array of person names managed in a round-robin fashion. The array has two indices, an In_Index denoting the index for the next input person name and an Out_Index denoting the index for the next output person name.

7.1/2 {*AI95-00433-01*} The Buffer is defined as an extension of the Synchronized_Queue interface (see 3.9.4), and as such promises to implement the abstraction defined by that interface. By doing so, the Buffer can be passed to the Transfer class-wide operation defined for objects of a type covered by Queue'Class.

7.2/5
```
{AI12-0178-1} type Person_Name_Array is array (Positive range <>)
   of Person_Name;  -- see 3.10.1
```

8/2
```
{AI95-00433-01} protected Buffer is new Synchronized_Queue with  -- see 3.9.4
   entry Append_Wait(Person : in Person_Name);
   entry Remove_First_Wait(Person : out Person_Name);
   function Cur_Count return Natural;
   function Max_Count return Natural;
   procedure Append(Person : in Person_Name);
   procedure Remove_First(Person : out Person_Name);
private
   Pool      : Person_Name_Array(1 .. 100);
   Count     : Natural := 0;
   In_Index, Out_Index : Positive := 1;
end Buffer;
```

9/2
```
{AI95-00433-01} protected body Buffer is
   entry Append_Wait(Person : in Person_Name)
      when Count < Pool'Length is
   begin
      Append(Person);
   end Append_Wait;
```

9.1/2
```
{AI95-00433-01}    procedure Append(Person : in Person_Name) is
   begin
      if Count = Pool'Length then
         raise Queue_Error with "Buffer Full";  -- see 11.3
      end if;
      Pool(In_Index) := Person;
      In_Index       := (In_Index mod Pool'Length) + 1;
      Count          := Count + 1;
   end Append;
```

10/2
```
{AI95-00433-01}    entry Remove_First_Wait(Person : out Person_Name)
      when Count > 0 is
   begin
      Remove_First(Person);
   end Remove_First_Wait;
```

11/2
```
{AI95-00433-01}    procedure Remove_First(Person : out Person_Name) is
   begin
      if Count = 0 then
         raise Queue_Error with "Buffer Empty"; -- see 11.3
      end if;
      Person    := Pool(Out_Index);
      Out_Index := (Out_Index mod Pool'Length) + 1;
      Count     := Count - 1;
   end Remove_First;
```

```
{AI95-00433-01}    function Cur_Count return Natural is                    12/2
   begin
       return Buffer.Count;
   end Cur_Count;
{AI95-00433-01}    function Max_Count return Natural is                    13/2
   begin
       return Pool'Length;
   end Max_Count;
end Buffer;
```

10 Program Structure and Compilation Issues

{*AI05-0299-1*} [The overall structure of programs and the facilities for separate compilation are described in this clause. A *program* is a set of *partitions*, each of which may execute in a separate address space, possibly on a separate computer. 1/3

Term entry: program — set of partitions, each of which can execute in a separate address space, possibly on a separate computer 1.a/5

Term entry: partition — part of a program, which consists of a set of interdependent library units 1.b/5
Note: Each partition can run in a separate address space, possibly on a separate computer. A program can contain just one partition, or it can be distributed across multiple partitions, which can execute concurrently.

As explained below, a partition is constructed from *library units*. Syntactically, the declaration of a library unit is a library_item, as is the body of a library unit. An implementation may support a concept of a *program library* (or simply, a "library"), which contains library_items and their subunits. Library units may be organized into a hierarchy of children, grandchildren, and so on.] 2

{*AI05-0299-1*} This clause has two subclauses: 10.1, "Separate Compilation" discusses compile-time issues related to separate compilation. 10.2, "Program Execution" discusses issues related to what is traditionally known as "link time" and "run time" — building and executing partitions. 3/3

Language Design Principles

We should avoid specifying details that are outside the domain of the language itself. The standard is intended (at least in part) to promote portability of Ada programs at the source level. It is not intended to standardize extra-language issues such as how one invokes the compiler (or other tools), how one's source is represented and organized, version management, the format of error messages, etc. 3.a

The rules of the language should be enforced even in the presence of separate compilation. Using separate compilation should not make a program less safe. 3.b

It should be possible to determine the legality of a compilation unit by looking only at the compilation unit itself and the compilation units upon which it depends semantically. As an example, it should be possible to analyze the legality of two compilation units in parallel if they do not depend semantically upon each other. 3.c

On the other hand, it may be necessary to look outside that set in order to generate code — this is generally true for generic instantiation and inlining, for example. Also on the other hand, it is generally necessary to look outside that set in order to check Post-Compilation Rules. 3.d

See also the "generic contract model" Language Design Principle of 12.3, "Generic Instantiation". 3.e

Wording Changes from Ada 83

{*AI05-0299-1*} The clause organization mentioned above is different from that of RM83. 3.f/3

10.1 Separate Compilation

[A *program unit* is either a package, a task unit, a protected unit, a protected entry, a generic unit, or an explicitly declared subprogram other than an enumeration literal. Certain kinds of program units can be separately compiled. Alternatively, they can appear physically nested within other program units. 1

Term entry: program unit — language construct that is a package, a task unit, a protected unit, a protected entry, a generic unit, or an explicitly declared subprogram other than an enumeration literal 1.a/5
Note: Certain kinds of program units can be separately compiled. Alternatively, they can appear physically nested within other program units.

The text of a program can be submitted to the compiler in one or more compilations. Each compilation is a succession of compilation_units. A compilation_unit contains either the declaration, the body, or a renaming of a program unit.] The representation for a compilation is implementation-defined. 2

Implementation defined: The representation for a compilation. 2.a

2.b **Ramification:** Some implementations might choose to make a compilation be a source (text) file. Others might allow multiple source files to be automatically concatenated to form a single compilation. Others still may represent the source in a nontextual form such as a parse tree. Note that the RM95 does not even define the concept of a source file.

2.c Note that a protected subprogram is a subprogram, and therefore a program unit. An instance of a generic unit is a program unit.

2.d A protected entry is a program unit, but protected entries cannot be separately compiled.

2.e/5 **Term entry: compilation unit** — program unit that is separately compiled
Note: A compilation_unit contains either the declaration, the body, or a renaming of a program unit.

3 A library unit is a separately compiled program unit, and is a package, subprogram, or generic unit. Library units may have other (logically nested) library units as children, and may have other program units physically nested within them. A root library unit, together with its children and grandchildren and so on, form a *subsystem*.

3.a/5 **Term entry: library unit** — separately compiled program unit, which is a package, a subprogram, or a generic unit
Note: Library units can have other (logically nested) library units as children, and can have other program units physically nested within them. A root library unit, together with its children and grandchildren and so on, form a subsystem.

Implementation Permissions

4 An implementation may impose implementation-defined restrictions on compilations that contain multiple compilation_units.

4.a **Implementation defined:** Any restrictions on compilations that contain multiple compilation_units.

4.b **Discussion:** For example, an implementation might disallow a compilation that contains two versions of the same compilation unit, or that contains the declarations for library packages P1 and P2, where P1 precedes P2 in the compilation but P1 has a with_clause that mentions P2.

Wording Changes from Ada 83

4.c The interactions between language issues and environmental issues are left open in Ada 95. The environment concept is new. In Ada 83, the concept of the program library, for example, appeared to be quite concrete, although the rules had no force, since implementations could get around them simply by defining various mappings from the concept of an Ada program library to whatever data structures were actually stored in support of separate compilation. Indeed, implementations were encouraged to do so.

4.d In RM83, it was unclear which was the official definition of "program unit". Definitions appeared in RM83-5, 6, 7, and 9, but not 12. Placing it here seems logical, since a program unit is sort of a potential compilation unit.

10.1.1 Compilation Units - Library Units

1 [A library_item is a compilation unit that is the declaration, body, or renaming of a library unit. Each library unit (except Standard) has a *parent unit*, which is a library package or generic library package.] A library unit is a *child* of its parent unit. The *root* library units are the children of the predefined library package Standard.

1.a **Ramification:** Standard is a library unit.

Syntax

2 compilation ::= {compilation_unit}

3 compilation_unit ::=
    context_clause library_item
  | context_clause subunit

4 library_item ::= [**private**] library_unit_declaration
  | library_unit_body
  | [**private**] library_unit_renaming_declaration

library_unit_declaration ::= 5
    subprogram_declaration | package_declaration
  | generic_declaration | generic_instantiation

library_unit_renaming_declaration ::= 6
  package_renaming_declaration
| generic_renaming_declaration
| subprogram_renaming_declaration

library_unit_body ::= subprogram_body | package_body 7

parent_unit_name ::= name 8

{*AI95-00397-01*} An overriding_indicator is not allowed in a subprogram_declaration, generic_instantiation, or subprogram_renaming_declaration that declares a library unit. 8.1/2

Reason: All of the listed items syntactically include overriding_indicator, but a library unit can never override anything. A majority of the ARG thought that allowing **not overriding** in that case would be confusing instead of helpful. 8.a.1/2

A *library unit* is a program unit that is declared by a library_item. When a program unit is a library unit, the prefix "library" is used to refer to it (or "generic library" if generic), as well as to its declaration and body, as in "library procedure", "library package_body", or "generic library package". The term *compilation unit* is used to refer to a compilation_unit. When the meaning is clear from context, the term is also used to refer to the library_item of a compilation_unit or to the proper_body of a subunit [(that is, the compilation_unit without the context_clause and the **separate** (parent_unit_name))]. 9

Discussion: In this example: 9.a

```
with Ada.Text_IO;                                                    9.b
package P is
   ...
end P;
```

the term "compilation unit" can refer to this text: "**with** Ada.Text_IO; **package** P **is** ... **end** P;" or to this text: "**package** P **is** ... **end** P;". We use this shorthand because it corresponds to common usage. 9.c

We like to use the word "unit" for declaration-plus-body things, and "item" for declaration or body separately (as in declarative_item). The terms "compilation_unit", "compilation unit", and "subunit" are exceptions to this rule. We considered changing "compilation_unit", "compilation unit" to "compilation_item", "compilation item", respectively, but we decided not to. 9.d

The *parent declaration* of a library_item (and of the library unit) is the declaration denoted by the parent_unit_name, if any, of the defining_program_unit_name of the library_item. If there is no parent_unit_name, the parent declaration is the declaration of Standard, the library_item is a *root* library_item, and the library unit (renaming) is a *root* library unit (renaming). The declaration and body of Standard itself have no parent declaration. The *parent unit* of a library_item or library unit is the library unit declared by its parent declaration. 10

Discussion: The declaration and body of Standard are presumed to exist from the beginning of time, as it were. There is no way to actually write them, since there is no syntactic way to indicate lack of a parent. An attempt to compile a package Standard would result in Standard.Standard. 10.a

Reason: Library units (other than Standard) have "parent declarations" and "parent units". Subunits have "parent bodies". We didn't bother to define the other possibilities: parent body of a library unit, parent declaration of a subunit, parent unit of a subunit. These are not needed, and might get in the way of a correct definition of "child". 10.b

[The children of a library unit occur immediately within the declarative region of the declaration of the library unit.] The *ancestors* of a library unit are itself, its parent, its parent's parent, and so on. [(Standard is an ancestor of every library unit.)] The *descendant* relation is the inverse of the ancestor relation. 11

Reason: These definitions are worded carefully to avoid defining subunits as children. Only library units can be children. 11.a

We use the unadorned term "ancestors" here to concisely define both "ancestor unit" and "ancestor declaration". 11.b

12 A library_unit_declaration or a library_unit_renaming_declaration is *private* if the declaration is immediately preceded by the reserved word **private**; it is otherwise *public*. A library unit is private or public according to its declaration. The *public descendants* of a library unit are the library unit itself, and the public descendants of its public children. Its other descendants are *private descendants*.

12.a **Discussion:** The first concept defined here is that a library_item is either public or private (not in relation to anything else — it's just a property of the library unit). The second concept is that a library_item is a public descendant or private descendant *of a given ancestor*. A given library_item can be a public descendant of one of its ancestors, but a private descendant of some other ancestor.

12.b A subprogram declared by a subprogram_body (as opposed to a subprogram_declaration) is always public, since the syntax rules disallow the reserved word **private** on a body.

12.c Note that a private library unit is a *public* descendant of itself, but a *private* descendant of its parent. This is because it is visible outside itself — its privateness means that it is not visible outside its parent.

12.d Private children of Standard are legal, and follow the normal rules. It is intended that implementations might have some method for taking an existing environment, and treating it as a package to be "imported" into another environment, treating children of Standard in the imported environment as children of the imported package.

12.e **Ramification:** Suppose we have a public library unit A, a private library unit A.B, and a public library unit A.B.C. A.B.C is a public descendant of itself and of A.B, but a private descendant of A; since A.B is private to A, we don't allow A.B.C to escape outside A either. This is similar to the situation that would occur with physical nesting, like this:

12.f
```
package A is
private
    package B is
        package C is
        end C;
    private
    end B;
end A;
```

12.g Here, A.B.C is visible outside itself and outside A.B, but not outside A. (Note that this example is intended to illustrate the visibility of program units from the outside; the visibility within child units is not quite identical to that of physically nested units, since child units are nested after their parent's declaration.)

12.1/2 {*AI95-00217-06*} For each library package_declaration in the environment, there is an implicit declaration of a *limited view* of that library package. The limited view of a package contains:

12.2/3 - {*AI95-00217-06*} {*AI05-0129-1*} {*AI05-0262-1*} For each package_declaration occurring immediately within the visible part, a declaration of the limited view of that package, with the same defining_program_unit_name.

12.3/3 - {*AI95-00217-06*} {*AI95-00326-01*} {*AI05-0108-1*} {*AI05-0129-1*} {*AI05-0262-1*} For each type_declaration occurring immediately within the visible part that is not an incomplete_type_declaration, an incomplete view of the type with no discriminant_part; if the type_declaration is tagged, then the view is a tagged incomplete view.

12.g.1/3 **Reason:** {*AI05-0108-1*} The incomplete view of a type does not have a discriminant_part even if the type_declaration does have one. This is necessary because semantic analysis (and the associated dependence on with_clauses) would be necessary to determine the types of the discriminants.

12.g.2/3 {*AI05-0129-1*} No incomplete views of incomplete types are included in the limited view. The rules of 3.10.1 ensure that the completion of any visible incomplete type is declared in the same visible part, so such an incomplete view would simply be redundant.

12.g.3/2 **Discussion:** {*AI95-00217-06*} The implementation model of a limited view is that it can be determined solely from the syntax of the source of the unit, without any semantic analysis. That allows it to be created without the semantic dependences of a full unit, which is necessary for it to break mutual dependences of units.

12.g.4/2 **Ramification:** The limited view does not include package instances and their contents. Semantic analysis of a unit (and dependence on its with_clauses) would be needed to determine the contents of an instance.

12.4/2 The limited view of a library package_declaration is private if that library package_declaration is immediately preceded by the reserved word **private**.

12.5/2 [There is no syntax for declaring limited views of packages, because they are always implicit.] The implicit declaration of a limited view of a library package [is not the declaration of a library unit (the

library package_declaration is); nonetheless, it] is a library_item. The implicit declaration of the limited view of a library package forms an (implicit) compilation unit whose context_clause is empty.

A library package_declaration is the completion of the declaration of its limited view. 12.6/2

To be honest: This is notwithstanding the rule in 3.11.1 that says that implicit declarations don't have completions. 12.h/2

Reason: This rule explains where to find the completions of the incomplete views defined by the limited view. 12.i/2

Legality Rules

The parent unit of a library_item shall be a [library] package or generic [library] package. 13

If a defining_program_unit_name of a given declaration or body has a parent_unit_name, then the given declaration or body shall be a library_item. The body of a program unit shall be a library_item if and only if the declaration of the program unit is a library_item. In a library_unit_renaming_declaration, the [(old)] name shall denote a library_item. 14

Discussion: We could have allowed nested program units to be children of other program units; their semantics would make sense. We disallow them to keep things simpler and because they wouldn't be particularly useful. 14.a

{*AI95-00217-06*} A parent_unit_name [(which can be used within a defining_program_unit_name of a library_item and in the **separate** clause of a subunit)], and each of its prefixes, shall not denote a renaming_declaration. [On the other hand, a name that denotes a library_unit_renaming_declaration is allowed in a nonlimited_with_clause and other places where the name of a library unit is allowed.] 15/2

If a library package is an instance of a generic package, then every child of the library package shall either be itself an instance or be a renaming of a library unit. 16

Discussion: A child of an instance of a given generic unit will often be an instance of a (generic) child of the given generic unit. This is not required, however. 16.a

Reason: Instances are forbidden from having noninstance children for two reasons: 16.b

1. We want all source code that can depend on information from the private part of a library unit to be inside the "subsystem" rooted at the library unit. If an instance of a generic unit were allowed to have a noninstance as a child, the source code of that child might depend on information from the private part of the generic unit, even though it is outside the subsystem rooted at the generic unit. 16.c
2. Disallowing noninstance children simplifies the description of the semantics of children of generic packages. 16.d

{*AI05-0004-1*} A child of a generic library package shall either be itself a generic unit or be a renaming of some other child of the same generic unit. 17/3

A child of a parent generic package shall be instantiated or renamed only within the declarative region of the parent generic. 18

{*AI95-00331-01*} For each child *C* of some parent generic package *P*, there is a corresponding declaration *C* nested immediately within each instance of *P*. For the purposes of this rule, if a child *C* itself has a child *D*, each corresponding declaration for *C* has a corresponding child *D*. [The corresponding declaration for a child within an instance is visible only within the scope of a with_clause that mentions the (original) child generic unit.] 19/2

Implementation Note: Within the child, like anything nested in a generic unit, one can make up-level references to the current instance of its parent, and thereby gain access to the formal parameters of the parent, to the types declared in the parent, etc. This “nesting” model applies even within the generic_formal_part of the child, as it does for a generic child of a nongeneric unit. 19.a

Ramification: Suppose P is a generic library package, and P.C is a generic child of P. P.C can be instantiated inside the declarative region of P. Outside P, P.C can be mentioned only in a with_clause. Conceptually, an instance I of P is a package that has a nested generic unit called I.C. Mentioning P.C in a with_clause allows I.C to be instantiated. I need not be a library unit, and the instantiation of I.C need not be a library unit. If I is a library unit, and an instance of I.C is a child of I, then this instance has to be called something other than C. 19.b

A library subprogram shall not override a primitive subprogram. 20

20.a **Reason:** This prevents certain obscure anomalies. For example, if a library subprogram were to override a subprogram declared in its parent package, then in a compilation unit that depends *in*directly on the library subprogram, the library subprogram could hide the overridden operation from all visibility, but the library subprogram itself would not be visible.

20.b Note that even without this rule, such subprograms would be illegal for tagged types, because of the freezing rules.

21 The defining name of a function that is a compilation unit shall not be an operator_symbol.

21.a **Reason:** Since overloading is not permitted among compilation units, it seems unlikely that it would be useful to define one as an operator. Note that a subunit could be renamed within its parent to be an operator.

Static Semantics

22 A subprogram_renaming_declaration that is a library_unit_renaming_declaration is a renaming-as-declaration, not a renaming-as-body.

23 [There are two kinds of dependences among compilation units:]

24/5 - {*AI12-0445-1*} [The *semantic dependences* (see below) are the ones necessary to check the compile-time rules across compilation unit boundaries; a compilation unit depends semantically on the other compilation units necessary to determine its legality. The visibility rules are based on the semantic dependences.

25 - The *elaboration dependences* (see 10.2) determine the order of elaboration of library_items.]

25.a **Discussion:** Don't confuse these kinds of dependences with the run-time dependences among tasks and masters defined in 9.3, “Task Dependence - Termination of Tasks”.

26/2 {*AI95-00217-06*} A library_item depends semantically upon its parent declaration. A subunit depends semantically upon its parent body. A library_unit_body depends semantically upon the corresponding library_unit_declaration, if any. The declaration of the limited view of a library package depends semantically upon the declaration of the limited view of its parent. The declaration of a library package depends semantically upon the declaration of its limited view. A compilation unit depends semantically upon each library_item mentioned in a with_clause of the compilation unit. In addition, if a given compilation unit contains an attribute_reference of a type defined in another compilation unit, then the given compilation unit depends semantically upon the other compilation unit. The semantic dependence relationship is transitive.

26.a **Discussion:** The “if any” in the third sentence is necessary because library subprograms are not required to have a subprogram_declaration.

26.b **To be honest:** If a given compilation unit contains a choice_parameter_specification, then the given compilation unit depends semantically upon the declaration of Ada.Exceptions.

26.c If a given compilation unit contains a pragma with an argument of a type defined in another compilation unit, then the given compilation unit depends semantically upon the other compilation unit.

26.d **Discussion:** For example, a compilation unit containing X'Address depends semantically upon the declaration of package System.

26.e For the Address attribute, this fixes a hole in Ada 83. Note that in almost all cases, the dependence will need to exist due to with_clauses, even without this rule. Hence, the rule has very little effect on programmers.

26.f Note that the semantic dependence does not have the same effect as a with_clause; in order to denote a declaration in one of those packages, a with_clause will generally be needed.

26.g Note that no special rule is needed for an attribute_definition_clause, since an expression after **use** will require semantic dependence upon the compilation unit containing the type_declaration of interest.

26.h/2 {*AI95-00217-06*} Unlike a full view of a package, a limited view does not depend semantically on units mentioned in with_clauses of the compilation_unit that defines the package. Formally, this is achieved by saying that the limited view has an empty context_clause. This is necessary so that they can be useful for their intended purpose: allowing mutual dependences between packages. The lack of semantic dependence limits the contents of a limited view to the items that can be determined solely from the syntax of the source of the package, without any semantic analysis. That allows it to be created without the semantic dependences of a full package.

Dynamic Semantics

{*AI95-00217-06*} The elaboration of the declaration of the limited view of a package has no effect. 26.1/2

NOTE 1 {*AI12-0442-1*} A simple program can consist of a single compilation unit. A compilation can have no compilation units; for example, its text can consist of pragmas. 27/5

Ramification: Such pragmas cannot have any arguments that are names, by a previous rule of this subclause. A compilation can even be entirely empty, which is probably not useful. 27.a

Some interesting properties of the three kinds of dependence: The elaboration dependences also include the semantic dependences, except that subunits are taken together with their parents. The semantic dependences partly determine the order in which the compilation units appear in the environment at compile time. At run time, the order is partly determined by the elaboration dependences. 27.b

The model whereby a child is inside its parent's declarative region, after the parent's declaration, as explained in 8.1, has the following ramifications: 27.c

- The restrictions on "early" use of a private type (RM83-7.4.1(4)) or a deferred constant (RM83-7.4.3(2)) do not apply to uses in child units, because they follow the full declaration. 27.d
- A library subprogram is never primitive, even if its profile includes a type declared immediately within the parent's package_specification, because the child is not declared immediately within the same package_specification as the type (so it doesn't declare a new primitive subprogram), and because the child is forbidden from overriding an old primitive subprogram. It is immediately within the same declarative region, but not the same package_specification. Thus, for a tagged type, it is not possible to call a child subprogram in a dispatching manner. (This is also forbidden by the freezing rules.) Similarly, it is not possible for the user to declare primitive subprograms of the types declared in the declaration of Standard, such as Integer (even if the rules were changed to allow a library unit whose name is an operator symbol). 27.e
- When the parent unit is "used" the simple names of the with'd child units are directly visible (see 8.4, "Use Clauses"). 27.f
- When a parent body with's its own child, the defining name of the child is directly visible, and the parent body is not allowed to include a declaration of a homograph of the child unit immediately within the declarative_part of the body (RM83-8.3(17)). 27.g

Note that "declaration of a library unit" is different from "library_unit_declaration" — the former includes subprogram_body. Also, we sometimes really mean "declaration of a view of a library unit", which includes library_unit_renaming_declarations. 27.h

The visibility rules generally imply that the renamed view of a library_unit_renaming_declaration has to be mentioned in a with_clause of the library_unit_renaming_declaration. 27.i

To be honest: The real rule is that the renamed library unit has to be visible in the library_unit_renaming_declaration. 27.j

Reason: In most cases, "has to be visible" means there has to be a with_clause. However, it is possible in obscure cases to avoid the need for a with_clause; in particular, a compilation unit such as "**package** P.Q **renames** P;" is legal with no with_clauses (though not particularly interesting). ASCII is physically nested in Standard, and so is not a library unit, and cannot be renamed as a library unit. 27.k

NOTE 2 The designator of a library function cannot be an operator_symbol, but a nonlibrary renaming_declaration is allowed to rename a library function as an operator. Within a partition, two library subprograms are required to have distinct names and hence cannot overload each other. However, renaming_declarations are allowed to define overloaded names for such subprograms, and a locally declared subprogram is allowed to overload a library subprogram. The expanded name Standard.L can be used to denote a root library unit L (unless the declaration of Standard is hidden) since root library unit declarations occur immediately within the declarative region of package Standard. 28

Examples

Examples of library units: 29

```
package Rational_Numbers.IO is   -- public child of Rational_Numbers, see 7.1
   procedure Put(R : in  Rational);
   procedure Get(R : out Rational);
end Rational_Numbers.IO;
```
30

```
private procedure Rational_Numbers.Reduce(R : in out Rational);
                                 -- private child of Rational_Numbers
```
31

```
with Rational_Numbers.Reduce;    -- refer to a private child
package body Rational_Numbers is
   ...
end Rational_Numbers;
```
32

33
```
with Rational_Numbers.IO; use Rational_Numbers;
with Ada.Text_io;                -- see A.10
procedure Main is                -- a root library procedure
   R : Rational;
begin
   R := 5/3;                     -- construct a rational number, see 7.1
   Ada.Text_IO.Put("The answer is: ");
   IO.Put(R);
   Ada.Text_IO.New_Line;
end Main;
```

34
```
with Rational_Numbers.IO;
package Rational_IO renames Rational_Numbers.IO;
                                 -- a library unit renaming declaration
```

35 Each of the above library_items can be submitted to the compiler separately.

35.a **Discussion:** *Example of a generic package with children:*

35.b
```
generic
   type Element is private;
   with function Image(E : Element) return String;
package Generic_Bags is
   type Bag is limited private; -- A bag of Elements.
   procedure Add(B : in out Bag; E : Element);
   function Bag_Image(B : Bag) return String;
private
   type Bag is ...;
end Generic_Bags;
```

35.c
```
generic
package Generic_Bags.Generic_Iterators is
   ... -- various additional operations on Bags.
```

35.d
```
   generic
      with procedure Use_Element(E : in Element);
         -- Called once per bag element.
   procedure Iterate(B : in Bag);
end Generic_Bags.Generic_Iterators;
```

35.e A package that instantiates the above generic units:

35.f
```
with Generic_Bags;
with Generic_Bags.Generic_Iterators;
package My_Abstraction is
    type My_Type is ...;
    function Image(X : My_Type) return String;
    package Bags_Of_My_Type is new Generic_Bags(My_Type, Image);
    package Iterators_Of_Bags_Of_My_Type is new Bags_Of_My_Type.Generic_Iterators;
end My_Abstraction;
```

35.g In the above example, Bags_Of_My_Type has a nested generic unit called Generic_Iterators. The second with_clause makes that nested unit visible.

35.h Here we show how the generic body could depend on one of its own children:

35.i
```
with Generic_Bags.Generic_Iterators;
package body Generic_Bags is
   procedure Add(B : in out Bag; E : Element) is ... end Add;
```

35.j
```
   package Iters is new Generic_Iterators;
```

35.k
```
   function Bag_Image(B : Bag) return String is
      Buffer : String(1..10_000);
      Last : Integer := 0;
```

35.l
```
      procedure Append_Image(E : in Element) is
         Im : constant String := Image(E);
      begin
         if Last /= 0 then -- Insert a comma.
            Last := Last + 1;
            Buffer(Last) := ',';
         end if;
         Buffer(Last+1 .. Last+Im'Length) := Im;
         Last := Last + Im'Length;
      end Append_Image;
```

```
      procedure Append_All is new Iters.Iterate(Append_Image);
   begin
      Append_All(B);
      return Buffer(1..Last);
   end Bag_Image;
end Generic_Bags;
```

35.m

Extensions to Ada 83

The syntax rule for library_item is modified to allow the reserved word **private** before a library_unit_declaration. 35.n

Children (other than children of Standard) are new in Ada 95. 35.o

Library unit renaming is new in Ada 95. 35.p

Wording Changes from Ada 83

Standard is considered a library unit in Ada 95. This simplifies the descriptions, since it implies that the parent of each library unit is a library unit. (Standard itself has no parent, of course.) As in Ada 83, the language does not define any way to recompile Standard, since the name given in the declaration of a library unit is always interpreted in relation to Standard. That is, an attempt to compile a package Standard would result in Standard.Standard. 35.q

Extensions to Ada 95

{*AI95-00217-06*} The concept of a limited view is new. Combined with limited_with_clauses (see 10.1.2), they facilitate construction of mutually recursive types in multiple packages. 35.r/2

Wording Changes from Ada 95

{*AI95-00331-01*} Clarified the wording so that a grandchild generic unit will work as expected. 35.s/2

Wording Changes from Ada 2005

{*AI05-0108-1*} {*AI05-0129-1*} **Correction:** Clarified the wording so that it is clear that limited views of types never have discriminants and never are of incomplete types. 35.t/3

10.1.2 Context Clauses - With Clauses

[A context_clause is used to specify the library_items whose names are needed within a compilation unit.] 1

Language Design Principles

The reader should be able to understand a context_clause without looking ahead. Similarly, when compiling a context_clause, the compiler should not have to look ahead at subsequent context_items, nor at the compilation unit to which the context_clause is attached. (We have not completely achieved this.) 1.a

{*AI95-00217-06*} A *ripple effect* occurs when the legality of a compilation unit could be affected by adding or removing an otherwise unneeded with_clause on some compilation unit on which the unit depends, directly or indirectly. We try to avoid ripple effects because they make understanding and maintenance more difficult. However, ripple effects can occur because of direct visibility (as in child units); this seems impossible to eliminate. The ripple effect for with_clauses is somewhat similar to the Beaujolais effect (see 8.4) for use_clauses, which we also try to avoid. 1.b/2

Syntax

context_clause ::= {context_item} 2

context_item ::= with_clause | use_clause 3

{*AI95-00217-06*} {*AI95-00262-01*} with_clause ::= limited_with_clause | nonlimited_with_clause 4/2

limited_with_clause ::= **limited** [**private**] **with** *library_unit*_name {, *library_unit*_name}; 4.1/2

nonlimited_with_clause ::= [**private**] **with** *library_unit*_name {, *library_unit*_name}; 4.2/2

Discussion: {*AI95-00217-06*} A limited_with_clause makes a limited view of a unit visible. 4.a/2

4.b/2 {*AI95-00262-01*} A with_clause containing the reserved word **private** is called a *private with_clause*. It can be thought of as making items visible only in the private part, although it really makes items visible everywhere except the visible part. It can be used both for documentation purposes (to say that a unit is not used in the visible part), and to allow access to private units that otherwise would be prohibited.

Name Resolution Rules

5 The *scope* of a with_clause that appears on a library_unit_declaration or library_unit_renaming_-declaration consists of the entire declarative region of the declaration[, which includes all children and subunits]. The scope of a with_clause that appears on a body consists of the body[, which includes all subunits].

5.a/2 **Discussion:** {*AI95-00262-01*} Suppose a nonprivate with_clause of a public library unit mentions one of its private siblings. (This is only allowed on the body of the public library unit.) We considered making the scope of that with_clause not include the visible part of the public library unit. (This would only matter for a subprogram_body, since those are the only kinds of body that have a visible part, and only if the subprogram_body completes a subprogram_declaration, since otherwise the with_clause would be illegal.) We did not put in such a rule for two reasons: (1) It would complicate the wording of the rules, because we would have to split each with_clause into pieces, in order to correctly handle "**with** P, Q;" where P is public and Q is private. (2) The conformance rules prevent any problems. It doesn't matter if a type name in the spec of the body denotes the completion of a private_type_declaration.

5.b A with_clause also affects visibility within subsequent use_clauses and pragmas of the same context_clause, even though those are not in the scope of the with_clause.

6/2 {*AI95-00217-06*} A library_item (and the corresponding library unit) is *named* in a with_clause if it is denoted by a *library_unit*_name in the with_clause. A library_item (and the corresponding library unit) is *mentioned* in a with_clause if it is named in the with_clause or if it is denoted by a prefix in the with_clause.

6.a **Discussion:** With_clauses control the visibility of declarations or renamings of library units. Mentioning a root library unit in a with_clause makes its declaration directly visible. Mentioning a nonroot library unit makes its declaration visible. See Clause 8 for details.

6.b/2 {*AI95-00114-01*} Note that this rule implies that "**with** A.B.C;" is almost equivalent to "**with** A, A.B, A.B.C;". The reason for making a with_clause apply to all the ancestor units is to avoid "visibility holes" — situations in which an inner program unit is visible while an outer one is not. Visibility holes would cause semantic complexity and implementation difficulty. (This is not exactly equivalent because the latter with_clause names A and A.B, while the previous one does not. Whether a unit is "named" does not have any effect on visibility, however, so it is equivalent for visibility purposes.)

7 [Outside its own declarative region, the declaration or renaming of a library unit can be visible only within the scope of a with_clause that mentions it. The visibility of the declaration or renaming of a library unit otherwise follows from its placement in the environment.]

Legality Rules

8/2 {*AI95-00262-01*} If a with_clause of a given compilation_unit mentions a private child of some library unit, then the given compilation_unit shall be one of:

9/2 - {*AI95-00262-01*} the declaration, body, or subunit of a private descendant of that library unit;

10/2 - {*AI95-00220-01*} {*AI95-00262-01*} the body or subunit of a public descendant of that library unit, but not a subprogram body acting as a subprogram declaration (see 10.1.4); or

11/2 - {*AI95-00262-01*} the declaration of a public descendant of that library unit, in which case the with_clause shall include the reserved word **private**.

11.a/2 **Reason:** {*AI95-00262-01*} The purpose of this rule is to prevent a private child from being visible from outside the subsystem rooted at its parent. A private child can be semantically depended-on without violating this principle if it is used in a private with_clause.

11.b **Discussion:** This rule violates the one-pass context_clauses Language Design Principle. We rationalize this by saying that at least that Language Design Principle works for legal compilation units.

Example: 11.c

```
package A is                                                    11.d
end A;

package A.B is                                                  11.e
end A.B;

private package A.B.C is                                        11.f
end A.B.C;

package A.B.C.D is                                              11.g
end A.B.C.D;

with A.B.C; -- (1)                                              11.h
private package A.B.X is
end A.B.X;

package A.B.Y is                                                11.i
end A.B.Y;

with A.B.C; -- (2)                                              11.j
package body A.B.Y is
end A.B.Y;

private with A.B.C; -- (3)                                      11.j.1/2
package A.B.Z is
end A.B.Z;
```

{*AI95-00262-01*} (1) is OK because it's a private child of A.B — it would be illegal if we made A.B.X a public child of A.B. (2) is OK because it's the body of a child of A.B. (3) is OK because it's a child of A.B, and it is a private with_clause. It would be illegal to say "**with** A.B.C;" on any library_item whose name does not start with "A.B". Note that mentioning A.B.C.D in a with_clause automatically mentions A.B.C as well, so "**with** A.B.C.D;" is illegal in the same places as "**with** A.B.C;". 11.k/2

{*AI05-0005-1*} {*AI95-00262-01*} {*AI95-00262-01*} {*AI05-0077-1*} {*AI05-0122-1*} {*AI12-0261-1*} A name denoting a library_item (or the corresponding declaration for a child of a generic within an instance — see 10.1.1), if it is visible only due to being mentioned in one or more with_clauses of a unit *U* that include the reserved word **private**, shall appear only within: 12/5

- a private part; 13/2
- {*AI12-0261-1*} a body of a public descendant of *U*, but not within the subprogram_specification of a body of a subprogram that is a public descendant of *U*; 14/5
- {*AI12-0261-1*} a private descendant of *U* or its body; or 15/5
- a pragma within a context clause. 16/2

Ramification: These rules apply only if all of the with_clauses that mention the name include the reserved word **private**. They do not apply if the name is mentioned in any with_clause that does not include **private**. 16.a/2

Reason: {*AI05-0077-1*} These rules make the library_item visible anywhere that is not visible outside the subsystem rooted at the compilation_unit having the private with_clause, including private parts of packages nested in the visible part, private parts of child packages, the visible part of private children, and context clause pragmas like Elaborate_All. 16.b/3

We considered having the scope of a private with_clause not include the visible part. However, that rule would mean that moving a declaration between the visible part and the private part could change its meaning from one legal interpretation to a different legal interpretation. For example: 16.c/2

```
package A is                                                    16.d/2
    function B return Integer;
end A;

function B return Integer;                                      16.e/2

with A;                                                         16.f/2
private with B;
package C is
    use A;
    V1 : Integer := B; -- (1)
private
    V2 : Integer := B; -- (2)
end C;
```

If we say that library subprogram B is not in scope in the visible part of C, then the B at (1) resolves to A.B, while (2) resolves to library unit B. Simply moving a declaration could silently change its meaning. With the legality rule defined above, the B at (1) is illegal. If the user really meant A.B, they still can say that. 16.g/2

17/2 {*AI95-00217-06*} [A library_item mentioned in a limited_with_clause shall be the implicit declaration of the limited view of a library package, not the declaration of a subprogram, generic unit, generic instance, or a renaming.]

17.a/2 **Proof:** This is redundant because only such implicit declarations are visible in a limited_with_clause. See 10.1.6.

18/2 {*AI95-00217-06*} {*AI95-00412-01*} A limited_with_clause shall not appear on a library_unit_body, subunit, or library_unit_renaming_declaration.

18.a/2 **Reason:** {*AI95-00412-01*} We don't allow a limited_with_clause on a library_unit_renaming_declaration because it would be useless and therefore probably is a mistake. A renaming cannot appear in a limited_with_clause (by the rule prior to this one), and a renaming of a limited view cannot appear in a nonlimited_with_clause (because the name would not be within the scope of a with_clause denoting the package, see 8.5.3). Nor could it be the parent of another unit. That doesn't leave anywhere that the name of such a renaming **could** appear, so we simply make writing it illegal.

19/2 {*AI95-00217-06*} A limited_with_clause that names a library package shall not appear:

20/3 - {*AI95-00217-06*} {*AI05-0040-1*} in the context_clause for the explicit declaration of the named library package or any of its descendants;

20.a/2 **Reason:** We have to explicitly disallow

20.b/2
```
limited with P;
package P is ...
```

20.c/2 as we can't depend on the semantic dependence rules to do it for us as with regular withs. This says “named” and not “mentioned” in order that

20.d/2
```
limited private with P.Child;
package P is ...
```

20.e/2 can be used to allow a mutual dependence between the private part of P and the private child P.Child, which occurs in interfacing and other problems. Since the child always semantically depends on the parent, this is the only way such a dependence can be broken.

20.f/3 {*AI05-0040-1*} The part about descendants catches examples like

20.g/3
```
limited with P;
package P.Child is ...
```

21/3 - {*AI95-00217-06*} {*AI05-0077-1*} {*AI05-0262-1*} within a context_clause for a library_item that is within the scope of a nonlimited_with_clause that mentions the same library package; or

21.a.1/3 **Ramification:** {*AI05-0077-1*} This applies to nonlimited_withclauses found in the same context_clause, as well as nonlimited_withclauses found on parent units.

21.a/3 **Reason:** {*AI05-0077-1*} Such a limited_with_clause could have no effect, and would be confusing. If a nonlimited_with_clause for the same package is inherited from a parent unit or given in the context_clause, the full view is available, which strictly provides more information than the limited view.

22/3 - {*AI95-00217-06*} {*AI05-0077-1*} {*AI05-0262-1*} within a context_clause for a library_item that is within the scope of a use_clause that names an entity declared within the declarative region of the library package.

22.a.1/3 **Ramification:** {*AI05-0077-1*} This applies to use_clauses found in the same context_clause, as well as use_clauses found in (or on) parent units.

22.a/2 **Reason:** This prevents visibility issues, where whether an entity is an incomplete or full view depends on how the name of the entity is written. The limited_with_clause cannot be useful, as we must have the full view available in the parent in order for the use_clause to be legal.

23/2 NOTE {*AI95-00217-06*} A library_item mentioned in a nonlimited_with_clause of a compilation unit is visible within the compilation unit and hence acts just like an ordinary declaration. Thus, within a compilation unit that mentions its declaration, the name of a library package can be given in use_clauses and can be used to form expanded names, a library subprogram can be called, and instances of a generic library unit can be declared. If a child of a parent generic package is mentioned in a nonlimited_with_clause, then the corresponding declaration nested within each visible instance is visible within the compilation unit. Similarly, a library_item mentioned in a limited_with_clause of a compilation unit is visible within the compilation unit and thus can be used to form expanded names.

23.a **Ramification:** The rules given for with_clauses are such that the same effect is obtained whether the name of a library unit is mentioned once or more than once by the applicable with_clauses, or even within a given with_clause.

If a with_clause mentions a library_unit_renaming_declaration, it only "mentions" the prefixes appearing explicitly in the with_clause (and the renamed view itself); the with_clause is not defined to mention the ancestors of the renamed entity. Thus, if X renames Y.Z, then "with X;" does not make the declarations of Y or Z visible. Note that this does not cause the dreaded visibility holes mentioned above. 23.b

Examples

{AI12-0429-1} Examples of use of with clauses, limited with clauses, and private with clauses: 24/5

```
{AI95-00433-01} package Office is                                                    25/2
end Office;

{AI95-00433-01} with Ada.Strings.Unbounded;                                          26/2
package Office.Locations is
   type Location is new Ada.Strings.Unbounded.Unbounded_String;
end Office.Locations;

{AI95-00433-01} limited with Office.Departments;   -- types are incomplete           27/2
private with Office.Locations;     -- only visible in private part
package Office.Employees is
   type Employee is private;

   function Dept_Of(Emp : Employee) return access Departments.Department;            28/2
   procedure Assign_Dept(Emp  : in out Employee;
                         Dept : access Departments.Department);

   ...                                                                               29/2
private
   type Employee is
      record
         Dept : access Departments.Department;
         Loc : Locations.Location;
         ...
      end record;
end Office.Employees;

{AI12-0178-1} limited with Office.Employees;                                         30/5
package Office.Departments is
   type Department is ...;

   function Manager_Of(Dept : Department) return access Employees.Employee;          31/2
   procedure Assign_Manager(Dept : in out Department;
                            Mgr  : access Employees.Employee);
   ...
end Office.Departments;
```

{AI95-00433-01} {AI12-0429-1} {AI12-0440-1} The limited_with_clause *can be used to support mutually dependent abstractions that are split across multiple packages. In this case, an employee is assigned to a department, and a department has a manager who is an employee. If a* with_clause *with the reserved word* ***private*** *appears on one library unit and mentions a second library unit, it provides visibility to the second library unit, but restricts that visibility to the private part and body of the first unit. The compiler checks that no use is made of the second unit in the visible part of the first unit.* 32/5

Extensions to Ada 83

The syntax rule for with_clause is modified to allow expanded name notation. 32.a

A use_clause in a context_clause may be for a package (or type) nested in a library package. 32.b

Wording Changes from Ada 83

The syntax rule for context_clause is modified to more closely reflect the semantics. The Ada 83 syntax rule implies that the use_clauses that appear immediately after a particular with_clause are somehow attached to that with_clause, which is not true. The new syntax allows a use_clause to appear first, but that is prevented by a textual rule that already exists in Ada 83. 32.c

The concept of "scope of a with_clause" (which is a region of text) replaces RM83's notion of "apply to" (a with_clause applies to a library_item) The visibility rules are interested in a region of text, not in a set of compilation units. 32.d

32.e No need to define “apply to” for use_clauses. Their semantics are fully covered by the “scope (of a use_clause)” definition in 8.4.

Incompatibilities With Ada 95

32.f/2 {*AI95-00220-01*} **Amendment Correction:** A subprogram body acting as a declaration cannot **with** a private child unit. This would allow public export of types declared in private child packages, and thus cannot be allowed. This was allowed by mistake in Ada 95; a subprogram that does this will now be illegal.

Extensions to Ada 95

32.g/2 {*AI95-00217-06*} limited_with_clauses are new. They make a limited view of a package visible, where all of the types in the package are incomplete. They facilitate construction of mutually recursive types in multiple packages.

32.h/3 {*AI95-00262-01*} {*AI05-0077-1*} The syntax rules for with_clause are modified to allow the reserved word **private**. Private with_clauses do not allow the use of their library_item in the visible part of their compilation_unit. They also allow using private units in more locations than in Ada 95.

Incompatibilities With Ada 2005

32.i/3 {*AI05-0040-1*} **Correction:** Added missing rule that a limited with clause cannot name an ancestor unit. This is incompatible if an Ada 2005 program does this, but as this is a new Ada 2005 feature and the unintentionally allowed capability is not useful, the incompatibility is very unlikely to occur in practice.

Wording Changes from Ada 2005

32.j/3 {*AI05-0077-1*} **Correction:** Fixed wording so that we are not checking whether something in a context_clause is “within the scope of” something, as context_clauses are never included in anything's scope. The intended meaning is unchanged, however.

32.k/3 {*AI05-0122-1*} **Correction:** Fixed wording so the rules for private with clauses also apply to "sprouted" generic child units.

Wording Changes from Ada 2012

32.l/5 {*AI12-0261-1*} **Correction:** Fixed wording so the privately withed units are visible in the subprogram specification of the body of a private descendant; otherwise, the private specification is legal but the completion is not.

10.1.3 Subunits of Compilation Units

1 [Subunits are like child units, with these (important) differences: subunits support the separate compilation of bodies only (not declarations); the parent contains a body_stub to indicate the existence and place of each of its subunits; declarations appearing in the parent's body can be visible within the subunits.]

Syntax

2
```
body_stub ::=
   subprogram_body_stub | package_body_stub
 | task_body_stub | protected_body_stub
```

3/3 {*AI95-00218-03*} {*AI05-0267-1*}
```
subprogram_body_stub ::=
   [overriding_indicator]
   subprogram_specification is separate
     [aspect_specification];
```

3.a **Discussion:** Although this syntax allows a parent_unit_name, that is disallowed by 10.1.1, “Compilation Units - Library Units”.

4/3 {*AI05-0267-1*}
```
package_body_stub ::=
  package body defining_identifier is separate
    [aspect_specification];
```

5/3 {*AI05-0267-1*}
```
task_body_stub ::=
```

task body defining_identifier **is separate**
[aspect_specification];

{*AI05-0267-1*} protected_body_stub ::= 6/3
protected body defining_identifier **is separate**
[aspect_specification];

subunit ::= **separate** (parent_unit_name) proper_body 7

Legality Rules

{*AI95-00243-01*} The *parent body* of a subunit is the body of the program unit denoted by its parent_unit_name. The term *subunit* is used to refer to a subunit and also to the proper_body of a subunit. The *subunits of a program unit* include any subunit that names that program unit as its parent, as well as any subunit that names such a subunit as its parent (recursively). 8/2

Reason: {*AI95-00243-01*} We want any rule that applies to a subunit to apply to a subunit of a subunit as well. 8.a.1/2

Term entry: subunit — body of a program unit that can be compiled separately from its enclosing program unit 8.a/5

The parent body of a subunit shall be present in the current environment, and shall contain a corresponding body_stub with the same defining_identifier as the subunit. 9

Discussion: This can't be a Name Resolution Rule, because a subunit is not a complete context. 9.a

{*AI05-0004-1*} A package_body_stub shall be the completion of a package_declaration or generic_package_declaration; a task_body_stub shall be the completion of a task declaration; a protected_body_stub shall be the completion of a protected declaration. 10/3

{*AI12-0444-1*} In contrast, a subprogram_body_stub can be defined without it being the completion of a previous declaration, [in which case the _stub declares the subprogram]. If the _stub is a completion, it shall be the completion of a subprogram_declaration or generic_subprogram_declaration. The profile of a subprogram_body_stub that completes a declaration shall conform fully to that of the declaration. 11/5

Discussion: The part about subprogram_body_stubs echoes the corresponding rule for subprogram_bodies in 6.3, "Subprogram Bodies". 11.a

A subunit that corresponds to a body_stub shall be of the same kind (package_, subprogram_, task_, or protected_) as the body_stub. The profile of a subprogram_body subunit shall be fully conformant to that of the corresponding body_stub. 12

A body_stub shall appear immediately within the declarative_part of a compilation unit body. This rule does not apply within an instance of a generic unit. 13

Discussion: This is a methodological restriction; that is, it is not necessary for the semantics of the language to make sense. 13.a

The defining_identifiers of all body_stubs that appear immediately within a particular declarative_part shall be distinct. 14

Post-Compilation Rules

For each body_stub, there shall be a subunit containing the corresponding proper_body. 15

NOTE The rules in 10.1.4, "The Compilation Process" say that a body_stub is equivalent to the corresponding proper_body. This implies: 16

- Visibility within a subunit is the visibility that would be obtained at the place of the corresponding body_stub (within the parent body) if the context_clause of the subunit were appended to that of the parent body. 17

  Ramification: Recursively. Note that this transformation might make the parent illegal; hence it is not a true equivalence, but applies only to visibility within the subunit. 17.a

- The effect of the elaboration of a body_stub is to elaborate the subunit. 18

18.a **Ramification:** The elaboration of a subunit is part of its parent body's elaboration, whereas the elaboration of a child unit is not part of its parent declaration's elaboration.

18.b **Ramification:** A library_item that is mentioned in a with_clause of a subunit can be hidden (from direct visibility) by a declaration (with the same identifier) given in the subunit. Moreover, such a library_item can even be hidden by a declaration given within the parent body since a library unit is declared in its parent's declarative region; this however does not affect the interpretation of the with_clauses themselves, since only library_items are visible or directly visible in with_clauses.

18.c The body of a protected operation cannot be a subunit. This follows from the syntax rules. The body of a protected unit can be a subunit.

Examples

19/5 *{AI12-0429-1} Example that defines package Parent without subunits:*

20
```
package Parent is
    procedure Inner;
end Parent;
```

21
```
with Ada.Text_IO;
package body Parent is
    Variable : String := "Hello, there.";
    procedure Inner is
    begin
        Ada.Text_IO.Put_Line(Variable);
    end Inner;
end Parent;
```

22/5 *{AI12-0429-1} {AI12-0440-1} Example showing how the body of procedure Inner can be turned into a subunit by rewriting the package body as follows (with the declaration of Parent remaining the same):*

23
```
package body Parent is
    Variable : String := "Hello, there.";
    procedure Inner is separate;
end Parent;
```

24
```
with Ada.Text_IO;
separate(Parent)
procedure Inner is
begin
    Ada.Text_IO.Put_Line(Variable);
end Inner;
```

Extensions to Ada 83

24.a Subunits of the same ancestor library unit are no longer restricted to have distinct identifiers. Instead, we require only that the full expanded names be distinct.

Extensions to Ada 95

24.b/2 *{AI95-00218-03}* An overriding_indicator (see 8.3.1) is allowed on a subprogram stub.

Wording Changes from Ada 95

24.c/2 *{AI95-00243-01}* Clarified that a subunit of a subunit is still a subunit.

Extensions to Ada 2005

24.d/3 *{AI05-0267-1}* An optional aspect_specification can be used in a body_stub. This is described in 13.1.1.

10.1.4 The Compilation Process

1 Each compilation unit submitted to the compiler is compiled in the context of an *environment* declarative_part (or simply, an *environment*), which is a conceptual declarative_part that forms the outermost declarative region of the context of any compilation. At run time, an environment forms the declarative_part of the body of the environment task of a partition (see 10.2, “Program Execution”).

Ramification: At compile time, there is no particular construct that the declarative region is considered to be nested within — the environment is the universe. 1.a

To be honest: The environment is really just a portion of a declarative_part, since there might, for example, be bodies that do not yet exist. 1.b

{*AI12-0445-1*} The declarative_items of the environment are library_items appearing in an order such that there are no forward semantic dependences. Each included subunit occurs in place of the corresponding stub. The visibility rules apply as if the environment were the outermost declarative region, except that with_clauses are necessary to make declarations of library units visible (see 10.1.2). 2/5

{*AI95-00217-06*} The mechanisms for creating an environment and for adding and replacing compilation units within an environment are implementation defined. The mechanisms for adding a compilation unit mentioned in a limited_with_clause to an environment are implementation defined. 3/2

Implementation defined: The mechanisms for creating an environment and for adding and replacing compilation units. 3.a

Implementation defined: The mechanisms for adding a compilation unit mentioned in a limited_with_clause to an environment. 3.a.1/2

Ramification: The traditional model, used by most Ada 83 implementations, is that one places a compilation unit in the environment by compiling it. Other models are possible. For example, an implementation might define the environment to be a directory; that is, the compilation units in the environment are all the compilation units in the source files contained in the directory. In this model, the mechanism for replacing a compilation unit with a new one is simply to edit the source file containing that compilation unit. 3.b

Name Resolution Rules

{*8652/0032*} {*AI95-00192-01*} {*AI05-0264-1*} If a library_unit_body that is a subprogram_body is submitted to the compiler, it is interpreted only as a completion if a library_unit_declaration with the same defining_program_unit_name already exists in the environment for a subprogram other than an instance of a generic subprogram or for a generic subprogram (even if the profile of the body is not type conformant with that of the declaration); otherwise, the subprogram_body is interpreted as both the declaration and body of a library subprogram. 4/3

Ramification: The principle here is that a subprogram_body should be interpreted as only a completion if and only if it "might" be legal as the completion of some preexisting declaration, where "might" is defined in a way that does not require overload resolution to determine. 4.a

Hence, if the preexisting declaration is a subprogram_declaration or generic_subprogram_declaration, we treat the new subprogram_body as its completion, because it "might" be legal. If it turns out that the profiles don't fully conform, it's an error. In all other cases (the preexisting declaration is a package or a generic package, or an instance of a generic subprogram, or a renaming, or a "spec-less" subprogram, or in the case where there is no preexisting thing), the subprogram_body declares a new subprogram. 4.b

See also AI83-00266/09. 4.c

Legality Rules

When a compilation unit is compiled, all compilation units upon which it depends semantically shall already exist in the environment; the set of these compilation units shall be *consistent* in the sense that the new compilation unit shall not semantically depend (directly or indirectly) on two different versions of the same compilation unit, nor on an earlier version of itself. 5

Discussion: For example, if package declarations A and B both say "**with** X;", and the user compiles a compilation unit that says "**with** A, B;", then the A and B have to be talking about the same version of X. 5.a

Ramification: What it means to be a "different version" is not specified by the language. In some implementations, it means that the compilation unit has been recompiled. In others, it means that the source of the compilation unit has been edited in some significant way. 5.b

Note that an implementation cannot require the existence of compilation units upon which the given one does not semantically depend. For example, an implementation is required to be able to compile a compilation unit that says "**with** A;" when A's body does not exist. It has to be able to detect errors without looking at A's body. 5.c

5.d/3 {*AI05-0229-1*} Similarly, the implementation has to be able to compile a call to a subprogram for which aspect Inline has been specified without seeing the body of that subprogram — inlining would not be achieved in this case, but the call is still legal.

5.e/3 {*AI95-00217-06*} {*AI05-0005-1*} The consistency rule applies to limited views as well as the full view of a compilation unit. That means that an implementation needs a way to enforce consistency of limited views, not just of full views.

Implementation Permissions

6/2 {*AI95-00217-06*} The implementation may require that a compilation unit be legal before it can be mentioned in a limited_with_clause or it can be inserted into the environment.

7/3 {*AI95-00214-01*} {*AI05-0229-1*} When a compilation unit that declares or renames a library unit is added to the environment, the implementation may remove from the environment any preexisting library_item or subunit with the same full expanded name. When a compilation unit that is a subunit or the body of a library unit is added to the environment, the implementation may remove from the environment any preexisting version of the same compilation unit. When a compilation unit that contains a body_stub is added to the environment, the implementation may remove any preexisting library_item or subunit with the same full expanded name as the body_stub. When a given compilation unit is removed from the environment, the implementation may also remove any compilation unit that depends semantically upon the given one. If the given compilation unit contains the body of a subprogram for which aspect Inline is True, the implementation may also remove any compilation unit containing a call to that subprogram.

7.a/3 **Ramification:** {*AI05-0005-1*} The permissions given in this paragraph correspond to the traditional model, where compilation units enter the environment by being compiled into it, and the compiler checks their legality at that time. An implementation model in which the environment consists of all source files in a given directory might not want to take advantage of these permissions. Compilation units would not be checked for legality as soon as they enter the environment; legality checking would happen later, when compilation units are compiled. In this model, compilation units might never be automatically removed from the environment; they would be removed when the user explicitly deletes a source file.

7.b Note that the rule is recursive: if the above permission is used to remove a compilation unit containing an inlined subprogram call, then compilation units that depend semantically upon the removed one may also be removed, and so on.

7.c Note that here we are talking about dependences among existing compilation units in the environment; it doesn't matter what with_clauses are attached to the new compilation unit that triggered all this.

7.d/3 {*AI05-0229-1*} An implementation may have other modes in which compilation units in addition to the ones mentioned above are removed. For example, an implementation might inline subprogram calls without an explicit aspect Inline. If so, it either has to have a mode in which that optimization is turned off, or it has to automatically regenerate code for the inlined calls without requiring the user to resubmit them to the compiler.

7.d.1/2 **Discussion:** {*8652/0108*} {*AI95-00077-01*} {*AI95-00114-01*} In the standard mode, implementations may only remove units from the environment for one of the reasons listed here, or in response to an explicit user command to modify the environment. It is not intended that the act of compiling a unit is one of the "mechanisms" for removing units other than those specified by this document.

7.e/2 {*AI95-00214-01*} These rules are intended to ensure that an implementation never need keep more than one compilation unit with any full expanded name. In particular, it is not necessary to be able to have a subunit and a child unit with the same name in the environment at one time.

8 NOTE 1 The rules of the language are enforced across compilation and compilation unit boundaries, just as they are enforced within a single compilation unit.

8.a/3 **Ramification:** {*AI05-0299-1*} Note that Clause 1 requires an implementation to detect illegal compilation units at compile time.

9/5 NOTE 2 {*AI12-0442-1*} An implementation can support a concept of a *library*, which contains library_items. If multiple libraries are supported, the implementation can document how a single environment is constructed when a compilation unit is submitted to the compiler. Naming conflicts between different libraries can, for example, be resolved by treating each library as the root of a hierarchy of child library units.

9.a **Implementation Note:** Alternatively, naming conflicts could be resolved via some sort of hiding rule.

Discussion: For example, the implementation might support a command to import library Y into library X. If a root library unit called LU (that is, Standard.LU) exists in Y, then from the point of view of library X, it could be called Y.LU. X might contain library units that say, "**with** Y.LU;". 9.b

NOTE 3 A compilation unit containing an instantiation of a separately compiled generic unit does not semantically depend on the body of the generic unit. Therefore, replacing the generic body in the environment does not result in the removal of the compilation unit containing the instantiation. 10

Implementation Note: Therefore, implementations have to be prepared to automatically instantiate generic bodies at link-time, as needed. This might imply a complete automatic recompilation, but it is the intent of the language that generic bodies can be (re)instantiated without forcing all of the compilation units that semantically depend on the compilation unit containing the instantiation to be recompiled. 10.a

Extensions to Ada 83

{*AI95-00077-01*} {*AI95-00114-01*} Ada 83 allowed implementations to require that the body of a generic unit be available when the instantiation is compiled; that permission is dropped in Ada 95. This isn't really an extension (it doesn't allow Ada users to write anything that they couldn't in Ada 83), but there isn't a more appropriate category, and it does allow users more flexibility when developing programs. 10.b/2

Wording Changes from Ada 95

{*8652/0032*} {*AI95-00192-01*} **Corrigendum:** The wording was clarified to ensure that a subprogram_body is not considered a completion of an instance of a generic subprogram. 10.c/2

{*AI95-00214-01*} The permissions to remove a unit from the environment were clarified to ensure that it is never necessary to keep multiple (sub)units with the same full expanded name in the environment. 10.d/2

{*AI95-00217-06*} Units mentioned in a limited_with_clause were added to several rules; limited views have the same presence in the environment as the corresponding full views. 10.e/2

10.1.5 Pragmas and Program Units

{*AI12-0417-1*} [This subclause discusses pragmas related to compilations.] 1/5

Paragraphs 2 through 7 were moved to Annex J, "Obsolescent Features".

Post-Compilation Rules

Certain pragmas are defined to be *configuration pragmas*; they shall appear before the first compilation_unit of a compilation. [They are generally used to select a partition-wide or system-wide option.] The pragma applies to all compilation_units appearing in the compilation, unless there are none, in which case it applies to all future compilation_units compiled into the same environment. 8

Implementation Permissions

{*AI95-00212-01*} An implementation may require that configuration pragmas that select partition-wide or system-wide options be compiled when the environment contains no library_items other than those of the predefined environment. In this case, the implementation shall still accept configuration pragmas in individual compilations that confirm the initially selected partition-wide or system-wide options. 9/2

This paragraph was deleted. 10.a/5

Paragraph 10 was moved to Annex J, "Obsolescent Features".

Wording Changes from Ada 95

This paragraph was deleted.{*8652/0033*} {*AI95-00136-01*} {*AI12-0417-1*} 10.b/5

This paragraph was deleted.{*8652/0034*} {*AI95-00041-01*} {*AI12-0417-1*} 10.c/5

This paragraph was deleted.{*AI12-0417-1*} 10.d/5

{*AI95-00212-01*} The permission to place restrictions was clarified to: 10.e/2

- Ensure that it applies only to partition-wide configuration pragmas, not ones like Assertion_Policy (see 11.4.2), which can be different in different units; and 10.f/2

10.g/2 • Ensure that confirming pragmas are always allowed.

Wording Changes from Ada 2005

10.h/5 *This paragraph was deleted.*{*AI05-0132-1*} {*AI12-0417-1*}

Wording Changes from Ada 2012

10.i/5 {*AI12-0417-1*} The terms “program unit pragma” and “library unit pragma” were moved to Annex J, “Obsolescent Features” (specifically to J.15) as all of the pragmas that use these terms are now in that annex.

10.1.6 Environment-Level Visibility Rules

1 [The normal visibility rules do not apply within a parent_unit_name or a context_clause, nor within a pragma that appears at the place of a compilation unit. The special visibility rules for those contexts are given here.]

Static Semantics

2/2 {*AI95-00217-06*} {*AI95-00312-01*} Within the parent_unit_name at the beginning of an explicit library_item, and within a nonlimited_with_clause, the only declarations that are visible are those that are explicit library_items of the environment, and the only declarations that are directly visible are those that are explicit root library_items of the environment. Within a limited_with_clause, the only declarations that are visible are those that are the implicit declaration of the limited view of a library package of the environment, and the only declarations that are directly visible are those that are the implicit declaration of the limited view of a root library package.

2.a **Ramification:** In “**package** P.Q.R **is** ... **end** P.Q.R;”, this rule requires P to be a root library unit, and Q to be a library unit (because those are the things that are directly visible and visible). Note that visibility does not apply between the “**end**” and the “;”.

2.b Physically nested declarations are not visible at these places.

2.c Although Standard is visible at these places, it is impossible to name it, since it is not directly visible, and it has no parent.

2.c.1/2 {*AI95-00217-06*} Only compilation units defining limited views can be mentioned in a limited_with_clause, while only compilation units defining full views (that is, the explicit declarations) can be mentioned in a nonlimited_with_clause. This resolves the conflict inherent in having two compilation units with the same defining name.

2.d/2 *This paragraph was deleted.*{*AI95-00312-01*}

3 Within a use_clause or pragma that is within a context_clause, each library_item mentioned in a previous with_clause of the same context_clause is visible, and each root library_item so mentioned is directly visible. In addition, within such a use_clause, if a given declaration is visible or directly visible, each declaration that occurs immediately within the given declaration's visible part is also visible. No other declarations are visible or directly visible.

3.a **Discussion:** Note the word “same”. For example, if a with_clause on a declaration mentions X, this does not make X visible in use_clauses and pragmas that are on the body. The reason for this rule is the one-pass context_clauses Language Design Principle.

3.b Note that the second part of the rule does not mention pragmas.

4 Within the parent_unit_name of a subunit, library_items are visible as they are in the parent_unit_name of a library_item; in addition, the declaration corresponding to each body_stub in the environment is also visible.

4.a **Ramification:** For a subprogram without a separate subprogram_declaration, the body_stub itself is the declaration.

5 Within a pragma that appears at the place of a compilation unit, the immediately preceding library_item and each of its ancestors is visible. The ancestor root library_item is directly visible.

{*AI95-00312-01*} Notwithstanding the rules of 4.1.3, an expanded name in a with_clause, a pragma in a context_clause, or a pragma that appears at the place of a compilation unit may consist of a prefix that denotes a generic package and a selector_name that denotes a child of that generic package. [(The child is necessarily a generic unit; see 10.1.1.)] 6/2

Reason: This rule allows **with** A.B; and **pragma** Elaborate(A.B); where A is a generic library package and B is one of its (generic) children. This is necessary because it is not normally legal to use an expanded name to reach inside a generic package. 6.a/2

Wording Changes from Ada 83

The special visibility rules that apply within a parent_unit_name or a context_clause, and within a pragma that appears at the place of a compilation_unit are clarified. 6.b

Note that a context_clause is not part of any declarative region. 6.c

We considered making the visibility rules within parent_unit_names and context_clauses follow from the context of compilation. However, this attempt failed for various reasons. For example, it would require use_clauses in context_clauses to be within the declarative region of Standard, which sounds suspiciously like a kludge. And we would still need a special rule to prevent seeing things (in our own context_clause) that were with-ed by our parent, etc. 6.d

Wording Changes from Ada 95

{*AI95-00217-06*} Added separate visibility rules for limited_with_clauses; the existing rules apply only to nonlimited_with_clauses. 6.e/2

{*AI95-00312-01*} Clarified that the name of a generic child unit may appear in a pragma in a context_clause. 6.f/2

10.2 Program Execution

An Ada *program* consists of a set of *partitions*[, which can execute in parallel with one another, possibly in a separate address space, and possibly on a separate computer.] 1

Post-Compilation Rules

{*AI12-0439-1*} {*AI12-0448-1*} A partition is a program or part of a program that can be invoked from outside the Ada implementation. [For example, on many systems, a partition can be an executable file generated by the system linker.] The user can *explicitly assign* library units to a partition. The assignment is done in an implementation-defined manner. The compilation units included in a partition are those of the explicitly assigned library units, as well as other compilation units *needed by* those library units. The compilation units needed by a given compilation unit (the *needed compilation units*) are determined as follows (unless specified otherwise via an implementation-defined pragma, or by some other implementation-defined means): 2/5

Discussion: From a run-time point of view, an Ada 95 partition is identical to an Ada 83 program — implementations were always allowed to provide inter-program communication mechanisms. The additional semantics of partitions is that interfaces between them can be defined to obey normal language rules (as is done in Annex E, “Distributed Systems”), whereas interfaces between separate programs had no particular semantics. 2.a

Implementation defined: The manner of explicitly assigning library units to a partition. 2.b

Implementation defined: The implementation-defined means, if any, of specifying which compilation units are needed by a given compilation unit. 2.c

Discussion: There are no pragmas that “specify otherwise” defined by the core language. However, an implementation is allowed to provide such pragmas, and in fact Annex E, “Distributed Systems” defines some pragmas whose semantics includes reducing the set of compilation units described here. 2.d

Term entry: needed compilation unit — compilation unit that is necessary to produce an executable partition, because some entity declared or defined within the unit is used elsewhere in the partition 2.e/5

- {*AI12-0448-1*} A compilation unit is a needed compilation unit of itself; 3/5

4/5 • {*AI12-0448-1*} If a compilation unit is among the needed compilation units, then so are any compilation units upon which it depends semantically;

5/5 • {*AI12-0448-1*} If a library_unit_declaration is among the needed compilation units, then so is any corresponding library_unit_body;

6/5 • {*AI95-00217-06*} {*AI12-0448-1*} If a compilation unit with stubs is among the needed compilation units, then so are any corresponding subunits;

6.a **Discussion:** Note that in the environment, the stubs are replaced with the corresponding proper_bodies.

6.1/5 • {*AI95-00217-06*} {*AI12-0448-1*} If the (implicit) declaration of the limited view of a library package is among the needed compilation units, then so is the explicit declaration of the library package.

6.b **Discussion:** Note that a child unit is not included just because its parent is included — to include a child, mention it in a with_clause.

6.c/2 {*AI95-00217-06*} A package is included in a partition even if the only reference to it is in a limited_with_clause. While this isn't strictly necessary (no objects of types imported from such a unit can be created), it ensures that all incomplete types are eventually completed, and is the least surprising option.

7 The user can optionally designate (in an implementation-defined manner) one subprogram as the *main subprogram* for the partition. A main subprogram, if specified, shall be a subprogram.

7.a **Discussion:** This may seem superfluous, since it follows from the definition. But we would like to have every error message that might be generated (before run time) by an implementation correspond to some explicitly stated “shall” rule.

7.b Of course, this does not mean that the “shall” rules correspond one-to-one with an implementation's error messages. For example, the rule that says overload resolution “shall” succeed in producing a single interpretation would correspond to many error messages in a good implementation — the implementation would want to explain to the user exactly why overload resolution failed. This is especially true for the syntax rules — they are considered part of overload resolution, but in most cases, one would expect an error message based on the particular syntax rule that was violated.

7.c **Implementation defined:** The manner of designating the main subprogram of a partition.

7.d **Ramification:** An implementation cannot require the user to specify, say, all of the library units to be included. It has to support, for example, perhaps the most typical case, where the user specifies just one library unit, the main program. The implementation has to do the work of tracking down all the other ones.

8 Each partition has an anonymous *environment task*[, which is an implicit outermost task whose execution elaborates the library_items of the environment declarative_part, and then calls the main subprogram, if there is one. A partition's execution is that of its tasks.]

8.a **Ramification:** An environment task has no master; all nonenvironment tasks have masters.

8.b An implementation is allowed to support multiple concurrent executions of the same partition.

9/5 {*AI12-0448-1*} [The order of elaboration of library units is determined primarily by the *elaboration dependences*.] There is an elaboration dependence of a given library_item upon another if the given library_item or any of its subunits depends semantically on the other library_item. In addition, if a given library_item or any of its subunits has a pragma Elaborate or Elaborate_All that names another library unit, then there is an elaboration dependence of the given library_item upon the body of the other library unit, and, for Elaborate_All only, upon each library_item that is a needed compilation unit of the declaration of the other library unit.

9.a.1/2 **Discussion:** {*8652/0107*} {*AI95-00180-01*} {*AI95-00256-01*} “Mentions” was used informally in the above rule; it was not intended to refer to the definition of *mentions* in 10.1.2. It was changed to “names” to make this clear.

9.a See above for a definition of which library_items are “needed by” a given declaration.

9.b/5 {*AI12-0417-1*} Note that elaboration dependences are among library_items, whereas the other two forms of dependence are among compilation units. Note that elaboration dependence includes semantic dependence. It's a little bit sad that the Elaborate_Body aspect can't be folded into this mechanism. It follows from the definition that the elaboration dependence relationship is transitive. Note that the wording of the rule does not need to take into account a

semantic dependence of a library_item or one of its subunits upon a subunit of a different library unit, because that can never happen.

The environment task for a partition has the following structure: 10

```
task Environment_Task;                                                    11
task body Environment_Task is                                             12
   ... (1) -- The environment declarative_part
           -- (that is, the sequence of library_items) goes here.
begin
   ... (2) -- Call the main subprogram, if there is one.
end Environment_Task;
```

Ramification: The name of the environment task is written in italics here to indicate that this task is anonymous. 12.a

Discussion: The model is different for a "passive partition" (see E.1). Either there is no environment task, or its sequence_of_statements is an infinite loop rather than a call on a main subprogram. 12.b

The environment declarative_part at (1) is a sequence of declarative_items consisting of copies of the library_items included in the partition[. The order of elaboration of library_items is the order in which they appear in the environment declarative_part]: 13

- The order of all included library_items is such that there are no forward elaboration dependences. 14

  Ramification: This rule is written so that if a library_item depends on itself, we don't require it to be elaborated before itself. See AI83-00113/12. This can happen only in pathological circumstances. For example, if a library subprogram_body has no corresponding subprogram_declaration, and one of the subunits of the subprogram_body mentions the subprogram_body in a with_clause, the subprogram_body will depend on itself. For another example, if a library_unit_body applies a pragma Elaborate_All to its own declaration, then the library_unit_body will depend on itself. 14.a

- {*AI05-0229-1*} {*AI12-0417-1*} Any included library_unit_declaration for which aspect Elaborate_Body is True is immediately followed by its library_unit_body, if included. 15/5

  Discussion: This implies that the body of such a library unit shall not "with" any of its own children, or anything else that depends semantically upon the declaration of the library unit. 15.a

- All library_items declared pure occur before any that are not declared pure. 16

- All preelaborated library_items occur before any that are not preelaborated. 17

  Discussion: Normally, if two partitions contain the same compilation unit, they each contain a separate *copy* of that compilation unit. See Annex E, "Distributed Systems" for cases where two partitions share the same copy of something. 17.a

  There is no requirement that the main subprogram be elaborated last. In fact, it is possible to write a partition in which the main subprogram cannot be elaborated last. 17.b

  Ramification: This declarative_part has the properties required of all environments (see 10.1.4). However, the environment declarative_part of a partition will typically contain fewer compilation units than the environment declarative_part used at compile time — only the "needed" ones are included in the partition. 17.c

There shall be a total order of the library_items that obeys the above rules. The order is otherwise implementation defined. 18

Discussion: {*AI12-0417-1*} The only way to violate this rule is to have Elaborate or Elaborate_All pragmas or Elaborate_Body aspects that cause circular ordering requirements, thus preventing an order that has no forward elaboration dependences. 18.a/5

Implementation defined: The order of elaboration of library_items. 18.b

To be honest: {*AI12-0005-1*} Notwithstanding what the RM says elsewhere, each rule that requires a declaration to have a corresponding completion is considered to be a Post-Compilation Rule when the declaration is that of a library unit. 18.c/5

Discussion: Such rules may be checked at "link time", for example. Rules requiring the completion to have certain properties, on the other hand, are checked at compile time of the completion. 18.d

The full expanded names of the library units and subunits included in a given partition shall be distinct. 19

Reason: This is a Post-Compilation Rule because making it a Legality Rule would violate the Language Design Principle labeled "legality determinable via semantic dependences". 19.a

20 The sequence_of_statements of the environment task (see (2) above) consists of either:

21 • A call to the main subprogram, if the partition has one. If the main subprogram has parameters, they are passed; where the actuals come from is implementation defined. What happens to the result of a main function is also implementation defined.

21.a **Implementation defined:** Parameter passing and function return for the main subprogram.

22 or:

23 • A null_statement, if there is no main subprogram.

23.a **Discussion:** For a passive partition, either there is no environment task, or its sequence_of_statements is an infinite loop. See E.1.

24/5 {*AI12-0439-1*} The mechanisms for building and running partitions are implementation defined. [These can be combined into one operation, as, for example, in dynamic linking, or "load-and-go" systems.]

24.a **Implementation defined:** The mechanisms for building and running partitions.

Dynamic Semantics

25 The execution of a program consists of the execution of a set of partitions. Further details are implementation defined. The execution of a partition starts with the execution of its environment task, ends when the environment task terminates, and includes the executions of all tasks of the partition. [The execution of the (implicit) task_body of the environment task acts as a master for all other tasks created as part of the execution of the partition. When the environment task completes (normally or abnormally), it waits for the termination of all such tasks, and then finalizes any remaining objects of the partition.]

25.a **Ramification:** The "further details" mentioned above include, for example, program termination — it is implementation defined. There is no need to define it here; it's entirely up to the implementation whether it wants to consider the program as a whole to exist beyond the existence of individual partitions.

25.b **Implementation defined:** The details of program execution, including program termination.

25.c **To be honest:** The execution of the partition terminates (normally or abnormally) when the environment task terminates (normally or abnormally, respectively).

Bounded (Run-Time) Errors

26 Once the environment task has awaited the termination of all other tasks of the partition, any further attempt to create a task (during finalization) is a bounded error, and may result in the raising of Program_Error either upon creation or activation of the task. If such a task is activated, it is not specified whether the task is awaited prior to termination of the environment task.

Implementation Requirements

27 The implementation shall ensure that all compilation units included in a partition are consistent with one another, and are legal according to the rules of the language.

27.a **Discussion:** The consistency requirement implies that a partition cannot contain two versions of the same compilation unit. That is, a partition cannot contain two different library units with the same full expanded name, nor two different bodies for the same program unit. For example, suppose we compile the following:

27.b
```
package A is -- Version 1.
    ...
end A;
```

27.c
```
with A;
package B is
end B;
```

27.d
```
package A is -- Version 2.
    ...
end A;
```

27.e
```
with A;
package C is
end C;
```

It would be wrong for a partition containing B and C to contain both versions of A. Typically, the implementation would require the use of Version 2 of A, which might require the recompilation of B. Alternatively, the implementation might automatically recompile B when the partition is built. A third alternative would be an incremental compiler that, when Version 2 of A is compiled, automatically patches the object code for B to reflect the changes to A (if there are any relevant changes — there might not be any). 27.f

An implementation that supported fancy version management might allow the use of Version 1 in some circumstances. In no case can the implementation allow the use of both versions in the same partition (unless, of course, it can prove that the two versions are semantically identical). 27.g

The core language says nothing about inter-partition consistency; see also Annex E, "Distributed Systems". 27.h

Implementation Permissions

{*AI05-0299-1*} The kind of partition described in this subclause is known as an *active* partition. An implementation is allowed to support other kinds of partitions, with implementation-defined semantics. 28/3

Implementation defined: The semantics of any nonactive partitions supported by the implementation. 28.a

Discussion: Annex E, "Distributed Systems" defines the concept of passive partitions; they may be thought of as a partition without an environment task, or as one with a particularly simple form of environment task, having an infinite loop rather than a call on a main subprogram as its sequence_of_statements. 28.b

An implementation may restrict the kinds of subprograms it supports as main subprograms. However, an implementation is required to support all main subprograms that are public parameterless library procedures. 29

Ramification: The implementation is required to support main subprograms that are procedures declared by generic_instantiations, as well as those that are children of library units other than Standard. Generic units are, of course, not allowed to be main subprograms, since they are not subprograms. 29.a

Note that renamings are irrelevant to this rule. This rules says which subprograms (not views) have to be supported. The implementation can choose any way it wants for the user to indicate which subprogram should be the main subprogram. An implementation might allow any name of any view, including those declared by renamings. Another implementation might require it to be the original name. Another implementation still might use the name of the source file or some such thing. 29.b

If the environment task completes abnormally, the implementation may abort any dependent tasks. 30

Reason: If the implementation does not take advantage of this permission, the normal action takes place — the environment task awaits those tasks. 30.a

The possibility of aborting them is not shown in the *Environment_Task* code above, because there is nowhere to put an exception_handler that can handle exceptions raised in both the environment declarative_part and the main subprogram, such that the dependent tasks can be aborted. If we put an exception_handler in the body of the environment task, then it won't handle exceptions that occur during elaboration of the environment declarative_part. If we were to move those things into a nested block_statement, with the exception_handler outside that, then the block_statement would await the library tasks we are trying to abort. 30.b

Furthermore, this is merely a permission, and is not fundamental to the model, so it is probably better to state it separately anyway. 30.c

Note that implementations (and tools like debuggers) can have modes that provide other behaviors in addition. 30.d

NOTE 1 {*AI12-0440-1*} An implementation can provide inter-partition communication mechanism(s) via special packages and pragmas. Standard pragmas for distribution and methods for specifying inter-partition communication are defined in Annex E, "Distributed Systems". If no such mechanisms are provided, then each partition is isolated from all others, and behaves as a program in and of itself. 31/5

Ramification: Not providing such mechanisms is equivalent to disallowing multi-partition programs. 31.a

An implementation may provide mechanisms to facilitate checking the consistency of library units elaborated in different partitions; Annex E, "Distributed Systems" does so. 31.b

NOTE 2 {*AI12-0440-1*} Partitions are not required to run in separate address spaces. For example, an implementation can support dynamic linking via the partition concept. 32/5

NOTE 3 {*AI12-0417-1*} {*AI12-0449-1*} An order of elaboration of library_items that is consistent with the partial ordering defined above does not always ensure that each library_unit_body is elaborated before any other compilation unit whose elaboration necessitates that the library_unit_body be already elaborated. (In particular, there is no requirement that the body of a library unit be elaborated as soon as possible after the library_unit_declaration is elaborated, unless the pragmas or aspects in 10.2.1 are used.) 33/5

34/5 NOTE 4 {*AI12-0442-1*} A partition (active or otherwise) does not necessarily have a main subprogram. In such a case, all the work done by the partition would be done by elaboration of various library_items, and by tasks created by that elaboration. Passive partitions, which cannot have main subprograms, are defined in Annex E, “Distributed Systems”.

34.a **Ramification:** The environment task is the outermost semantic level defined by the language.

34.b Standard has no private part. This prevents strange implementation-dependences involving private children of Standard having visibility upon Standard's private part. It doesn't matter where the body of Standard appears in the environment, since it doesn't do anything. See Annex A, “Predefined Language Environment”.

34.c Note that elaboration dependence is carefully defined in such a way that if (say) the body of something doesn't exist yet, then there is no elaboration dependence upon the nonexistent body. (This follows from the fact that “needed by” is defined that way, and the elaboration dependences caused by a pragma Elaborate or Elaborate_All are defined in terms of “needed by”.) This property allows us to use the environment concept both at compile time and at partition-construction time/run time.

Extensions to Ada 83

34.d The concept of partitions is new to Ada 95.

34.e A main subprogram is now optional. The language-defined restrictions on main subprograms are relaxed.

Wording Changes from Ada 83

34.f Ada 95 uses the term “main subprogram” instead of Ada 83's “main program” (which was inherited from Pascal). This is done to avoid confusion — a main subprogram is a subprogram, not a program. The program as a whole is an entirely different thing.

Wording Changes from Ada 95

34.g/2 {*AI95-00256-01*} The mistaken use of “mentions” in the elaboration dependence rule was fixed.

34.h/5 {*AI95-00217-06*} {*AI12-0448-1*} The *needed* relationship was extended to include limited views.

10.2.1 Elaboration Control

1/5 {*AI12-0417-1*} [This subclause defines aspects and pragmas that help control the elaboration order of library_items.]

Language Design Principles

1.a The rules governing preelaboration are designed to allow it to be done largely by bulk initialization of statically allocated storage from information in a “load module” created by a linker. Some implementations may require run-time code to be executed in some cases, but we consider these cases rare enough that we need not further complicate the rules.

1.b It is important that programs be able to declare data structures that are link-time initialized with aggregates, string_literals, and concatenations thereof. It is important to be able to write link-time evaluated expressions involving the First, Last, and Length attributes of such data structures (including variables), because they might be initialized with positional aggregates or string_literals, and we don't want the user to have to count the elements. There is no corresponding need for accessing discriminants, since they can be initialized with a static constant, and then the constant can be referred to elsewhere. It is important to allow link-time initialized data structures involving discriminant-dependent components. It is important to be able to write link-time evaluated expressions involving pointers (both access values and addresses) to the above-mentioned data structures.

1.c The rules also ensure that no Elaboration_Check need be performed for calls on library-level subprograms declared within a preelaborated package. This is true also of the Elaboration_Check on task activation for library level task types declared in a preelaborated package. However, it is not true of the Elaboration_Check on instantiations.

1.d A static expression should never prevent a library unit from being preelaborable.

Paragraphs 2 through 4 were moved to Annex J, “Obsolescent Features”.

Legality Rules

5 An elaborable construct is preelaborable unless its elaboration performs any of the following actions:

5.a **Ramification:** A *preelaborable* construct can be elaborated without using any information that is available only at run time. Note that we don't try to prevent exceptions in preelaborable constructs; if the implementation wishes to generate code to raise an exception, that's OK.

Because there is no flow of control and there are no calls (other than to predefined subprograms), these run-time properties can actually be detected at compile time. This is necessary in order to require compile-time enforcement of the rules. 5.b

- The execution of a statement other than a null_statement. 6

  Ramification: A preelaborable construct can contain labels and null_statements. 6.a

- {*AI12-0175-1*} A call to a subprogram other than: 7/5
  - a static function; 7.1/5
  - an instance of Unchecked_Conversion (see 13.9); 7.2/5
  - a function declared in System.Storage_Elements (see 13.7.1); or 7.3/5
  - the functions To_Pointer and To_Address declared in an instance of System.Address_to_Access_Conversions (see 13.7.2). 7.4/5

    Ramification: {*AI12-0175-1*} The parameters of any such function have to pass the preelaborability rules, so they typically have to be static expressions. The extra allowed functions are all forms of conversion that the compiler understands, so there should be little implementation burden. Note that such a call might raise an exception; preelaborable is not the same as elaborable with no code. 7.a/5

- The evaluation of a primary that is a name of an object, unless the name is a static expression, or statically denotes a discriminant of an enclosing type. 8

  Ramification: One can evaluate such a name, but not as a primary. For example, one can evaluate an attribute of the object. One can evaluate an attribute_reference, so long as it does not denote an object, and its prefix does not disobey any of these rules. For example, Obj'Access, Obj'Unchecked_Access, and Obj'Address are generally legal in preelaborated library units. 8.a

- {*AI95-00161-01*} {*AI05-0028-1*} The creation of an object [(including a component)] that is initialized by default, if its type does not have preelaborable initialization. Similarly, the evaluation of an extension_aggregate with an ancestor subtype_mark denoting a subtype of such a type. 9/3

  Ramification: One can declare these kinds of types, but one cannot create objects of those types. 9.a

  It is also nonpreelaborable to create an object if that will cause the evaluation of a default expression that will call a user-defined function. This follows from the rule above forbidding nonnull statements. 9.b

  This paragraph was deleted.{*AI95-00161-01*} 9.c/2

- {*AI12-0232-1*} The elaboration of any elaborable construct that is not preelaborable. 9.1/5

{*AI12-0232-1*} A generic declaration is preelaborable unless every instance would perform one of the above actions. 9.2/5

Ramification: {*AI12-0232-1*} {*AI12-0417-1*} A generic declaration is preelaborable unless there is no instance that could be declared preelaborated. For instance, a generic package declaration that directly contains a variable initialized by a non-static function that is not a formal function is not preelaborable (and thus would be illegal if the Preelaborate aspect were applied to it). 9.d/5

{*AI95-00403-01*} A generic body is preelaborable only if elaboration of a corresponding instance body would not perform any such actions, presuming that: 10/2

- {*AI95-00403-01*} {*AI05-0028-1*} {*AI12-0417-1*} the actual for each discriminated formal derived type, formal private type, or formal private extension declared within the formal part of the generic unit is a type that does not have preelaborable initialization, unless the Preelaborable_Initialization aspect was specified for the formal type; 10.1/5
- {*AI95-00403-01*} the actual for each formal type is nonstatic; 10.2/2
- {*AI95-00403-01*} the actual for each formal object is nonstatic; and 10.3/2
- {*AI95-00403-01*} the actual for each formal subprogram is a user-defined subprogram. 10.4/2

  Discussion: {*AI95-00403-01*} This is an “assume-the-worst” rule. The elaboration of a generic unit doesn't perform any of the actions listed above, because its sole effect is to establish that the generic can from now on be instantiated. 10.a.1/2

So the elaboration of the generic itself is not the interesting part when it comes to preelaboration rules. The interesting part is what happens when you elaborate "any instantiation" of the generic. For instance, declaring an object of a limited formal private type might well start tasks, call functions, and do all sorts of nonpreelaborable things. We prevent these situations by assuming that the actual parameters are as badly behaved as possible.

10.a **Reason:** Without this rule about generics, we would have to forbid instantiations in preelaborated library units, which would significantly reduce their usefulness.

11/5 {*8652/0035*} {*AI95-00002-01*} {*AI05-0034-1*} {*AI05-0243-1*} {*AI12-0417-1*} When the library unit aspect (see 13.1.1) Preelaborate of a program unit is True, the unit is said to be *preelaborated*. When the Preelaborate aspect is specified True for a library unit, all compilation units of the library unit are preelaborated. The declaration and body of a preelaborated library unit, and all subunits that are elaborated as part of elaborating the library unit, shall be preelaborable. All compilation units of a preelaborated library unit shall depend semantically only on declared pure or preelaborated library_items. In addition to the places where Legality Rules normally apply (see 12.3), these rules also apply in the private part of an instance of a generic unit. [If a library unit is preelaborated, then its declaration, if any, and body, if any, are elaborated prior to all nonpreelaborated library_items of the partition.]

11.a **Ramification:** In a generic body, we assume the worst about formal private types and extensions.

11.a.1/1 {*8652/0035*} {*AI95-00002-01*} Subunits of a preelaborated subprogram unit do not need to be preelaborable. This is needed in order to be consistent with units nested in a subprogram body, which do not need to be preelaborable even if the subprogram is preelaborated. However, such subunits cannot depend semantically on nonpreelaborated units, which is also consistent with nested units.

11.a.2/5 {*AI12-0417-1*} If a unit is not preelaborated, the value of the Preelaborate aspect for that unit is False.

11.b/3 **Aspect Description for Preelaborate:** Code execution during elaboration is avoided for a given package.

11.c/5 **Discussion:** {*AI12-0417-1*} Rules for specifying a library unit aspect (like Preelaborate) are found in 13.1.1.

11.1/5 {*AI95-00161-01*} {*AI12-0409-1*} The following rules specify which entities have *preelaborable initialization*, namely that the Preelaborable_Initialization aspect of the entity is True:

11.2/5 - {*AI05-0028-1*} {*AI12-0409-1*} {*AI12-0444-1*} The partial view of a private type or private extension, a protected type without entry_declarations, a generic formal private type, or a generic formal derived type, has preelaborable initialization if and only if the Preelaborable_Initialization aspect has been specified True for them. [A protected type with entry_declarations or a task type never has preelaborable initialization.] The Preelaborable_Initialization aspect of a partial view of a type may be specified as False, even if the full view of the type has preelaborable initialization. Similarly, a generic formal type may be specified with Preelaborable_Initialization False, even if the actual type in an instance has preelaborable initialization.

11.3/2 - A component (including a discriminant) of a record or protected type has preelaborable initialization if its declaration includes a default_expression whose execution does not perform any actions prohibited in preelaborable constructs as described above, or if its declaration does not include a default expression and its type has preelaborable initialization.

11.4/3 - {*AI05-0028-1*} {*AI05-0221-1*} A derived type has preelaborable initialization if its parent type has preelaborable initialization and if the noninherited components all have preelaborable initialization. However, a controlled type with an Initialize procedure that is not a null procedure does not have preelaborable initialization.

11.5/2 - {*AI95-00161-01*} {*AI95-00345-01*} A view of a type has preelaborable initialization if it is an elementary type, an array type whose component type has preelaborable initialization, a record type whose components all have preelaborable initialization, or an interface type.

11.6/5 {*AI95-00161-01*} {*AI12-0409-1*} The following attribute is defined for a nonformal composite subtype S declared within the visible part of a package or a generic package, or a generic formal private subtype or formal derived subtype:

S'Preelaborable_Initialization 11.7/5

{*AI12-0409-1*} This attribute is of Boolean type, and its value reflects whether the type of S has preelaborable initialization. The value of this attribute, the type-related Preelaborable_Initialization aspect, may be specified for any type for which the attribute is defined. The value shall be specified by a static expression, unless the type is not a formal type but is nevertheless declared within a generic package. In this latter case, the value may also be specified by references to the Preelaborable_Initialization attribute of one or more formal types visible at the point of the declaration of the composite type, conjoined with **and**.

Ramification: {*AI12-0409-1*} "Formal types visible at the point of the declaration" includes all visible formal types, including those that might have been declared in formal packages or in child packages. 11.d/5

Aspect Description for Preelaborable_Initialization: Declares that a type has preelaborable initialization. 11.e/5

{*AI95-00161-01*} {*AI95-00345-01*} {*AI05-0028-1*} {*AI12-0409-1*} If the Preelaborable_Initialization aspect is specified True for a private type or a private extension, the full view of the type shall have preelaborable initialization. If the aspect is specified True for a protected type, the protected type shall not have entries, and each component of the protected type shall have preelaborable initialization. If the aspect is specified True for a generic formal type, then in a generic_instantiation the corresponding actual type shall have preelaborable initialization. If the aspect definition includes one or more Preelaborable_Initialization attribute_references, then the full view of the type shall have preelaborable initialization presuming the types mentioned in the prefixes of the attribute_references all have preelaborable initialization. For any other composite type, the aspect shall be specified statically True or False only if it is confirming. In addition to the places where Legality Rules normally apply (see 12.3), these rules apply also in the private part of an instance of a generic unit. 11.8/5

Reason: {*AI05-0028-1*} {*AI12-0409-1*} The reason why we need to be allowed to specify the aspect for private types, private extensions, and protected types is fairly clear: the properties of the full view determine whether the type has preelaborable initialization or not; in order to preserve privacy we need a way to express on the partial view that the full view is well-behaved. The reason why we need to be allowed to specify the aspect for other composite types is more subtle: a nonnull override for Initialize might occur in the private part, even for a nonprivate type; in order to preserve privacy, we need a way to express on a type declared in a visible part that the private part does not contain any nasty override of Initialize. 11.f/5

Ramification: {*AI12-0409-1*} Not only do protected types with entry_declarations and task types not have preelaborable initialization, but they cannot have Preelaborable_Initialization aspect specified True for them. 11.g/5

Implementation Advice

In an implementation, a type declared in a preelaborated package should have the same representation in every elaboration of a given version of the package, whether the elaborations occur in distinct executions of the same program, or in executions of distinct programs or partitions that include the given version. 12

Implementation Advice: A type declared in a preelaborated package should have the same representation in every elaboration of a given version of the package. 12.a/2

Paragraphs 13 through 15 were moved to Annex J, "Obsolescent Features".

Static Semantics

{*AI95-00366-01*} {*AI05-0035-1*} {*AI12-0232-1*} A *pure* program unit is a preelaborable program unit whose elaboration does not perform any of the following actions: 15.1/5

- the elaboration of a variable declaration; 15.2/2
- the evaluation of an allocator of an access-to-variable type; for the purposes of this rule, the partial view of a type is presumed to have nonvisible components whose default initialization evaluates such an allocator; 15.3/2

Reason: {*AI05-0004-1*} Such an allocator would provide a backdoor way to get a global variable into a pure unit, so it is prohibited. Most such allocators are illegal anyway, as their type is required to have Storage_Size = 0 (see the next two rules). But access parameters and access discriminants don't necessarily disallow allocators. However, a call is 15.a.1/3

also illegal here (by the preelaboration rules), so access parameters cannot cause trouble. So this rule is really about prohibiting allocators in discriminant constraints:

15.a.2/3
```
type Rec (Acc : access Integer) is record
    C : Character;
end record;
```

15.a.3/3
```
Not_Const : constant Rec (Acc => new Integer'(2)); -- Illegal in a pure unit.
```

15.a/5 {*AI05-0004-1*} {*AI12-0417-1*} The second half of the rule is needed because aggregates can specify the default initialization of a private type or extension using <> or the ancestor subtype of an extension aggregate. The subtype of a component could use an allocator to initialize an access discriminant; the type still could have the Preelaborable_Initialization aspect specified. Ada 95 did not allow such private types to have preelaborable initialization, so such a default initialization could not have occurred. Thus this rule is not incompatible with Ada 95.

15.4/3 • {*AI05-0035-1*} the elaboration of the declaration of a nonderived named access-to-variable type unless the Storage_Size of the type has been specified by a static expression with value zero or is defined by the language to be zero;

15.b/2 **Discussion:** A remote access-to-class-wide type (see E.2.2) has its Storage_Size defined to be zero.

15.c/2 **Reason:** {*AI95-00366-01*} We disallow most named access-to-object types because an allocator has a side effect; the pool constitutes variable data. We allow access-to-subprogram types because they don't have allocators. We even allow named access-to-object types if they have an empty predefined pool (they can't have a user-defined pool as System.Storage_Pools is not pure). In this case, most attempts to use an allocator are illegal, and any others (in a generic body) will raise Storage_Error.

15.5/5 • {*AI05-0035-1*} {*AI12-0232-1*} the elaboration of the declaration of a nonderived named access-to-constant type for which the Storage_Size has been specified by an expression other than a static expression with value zero;

15.d/2 **Discussion:** We allow access-to-constant types so long as there is no user-specified nonzero Storage_Size; if there were a user-specified nonzero Storage_Size restricting the size of the storage pool, allocators would be problematic since the package is supposedly ‘stateless’, and the allocated size count for the storage pool would represent state.

15.6/5 • {*AI12-0232-1*} the elaboration of any program unit that is not pure.

15.7/5 {*AI12-0232-1*} A generic declaration is pure unless every instance would perform one of the above actions.

15.d.1/5 **Ramification:** {*AI12-0232-1*} {*AI12-0417-1*} A generic declaration is pure unless there is no instance that could be declared pure. For instance, a generic package declaration that directly contains a variable declaration is not pure (and thus would be illegal if aspect Pure is specified for it).

15.8/5 {*AI05-0035-1*} {*AI12-0232-1*} A generic body is pure only if elaboration of a corresponding instance body would not perform any such actions presuming any composite formal types have nonvisible components whose default initialization evaluates an allocator of an access-to-variable type.

15.9/5 {*AI95-00366-01*} {*AI12-0232-1*} The Storage_Size for an anonymous access-to-variable type declared at library level in a library unit that is declared pure is defined to be zero.

15.e/2 **Ramification:** This makes allocators illegal for such types (see 4.8), making a storage pool unnecessary for these types. A storage pool would represent state.

15.f/2 Note that access discriminants and access parameters are never library-level, even when they are declared in a type or subprogram declared at library-level. That's because they have their own special accessibility rules (see 3.10.2).

Legality Rules

16/2 *This paragraph was deleted.*{*AI95-00366-01*}

17/5 {*AI95-00366-01*} {*AI05-0034-1*} {*AI05-0035-1*} {*AI05-0243-1*} {*AI12-0417-1*} When the library unit aspect Pure of a program unit is True, the unit is said to be *declared pure*. When the Pure aspect is specified True for a library unit, all compilation units of the library unit are declared pure. In addition, the limited view of any library package is declared pure. The declaration and body of a declared pure library unit, and all subunits that are elaborated as part of elaborating the library unit, shall be pure. All compilation units of a declared pure library unit shall depend semantically only on declared pure

library_items. In addition to the places where Legality Rules normally apply (see 12.3), these rules also apply in the private part of an instance of a generic unit. Furthermore, the full view of any partial view declared in the visible part of a declared pure library unit that has any available stream attributes shall support external streaming (see 13.13.2).

This paragraph was deleted.{*AI05-0243-1*} 17.a/3

Discussion: A declared-pure package is useful for defining types to be shared between partitions with no common address space. 17.b

Reason: Note that generic packages are not mentioned in the list of things that can contain variable declarations. Note that the Ada 95 rules for deferred constants make them allowable in library units that are declared pure; that isn't true of Ada 83's deferred constants. 17.c

Ramification: {*AI95-00366-01*} Anonymous access types are allowed. 17.d/2

{*AI05-0243-1*} A limited view is not a library unit, so any rule that starts "declared pure library unit" does not apply to a limited view. In particular, the 3rd and last sentences never apply to limited views. However, a limited view is a library_item, so rules that discuss "declared pure library_items" do include limited views. 17.d.1/3

{*AI12-0417-1*} If a unit is not declared pure, the value of the Pure aspect for that unit is False. 17.d.2/5

Reason: {*AI95-00366-01*} Ada 95 didn't allow any access types as these (including access-to-subprogram types) cause trouble for Annex E, "Distributed Systems", because such types allow access values in a shared passive partition to designate objects in an active partition, thus allowing inter-address space references. We decided to disallow such uses in the relatively rare cases where they cause problems, rather than making life harder for the majority of users. Types declared in a pure package can be used in remote operations only if they are externally streamable. That simply means that there is a means to transport values of the type; that's automatically true for nonlimited types that don't have an access part. The only tricky part about this is to avoid privacy leakage; that was handled by ensuring that any private types (and private extensions) declared in a pure package that have available stream attributes (which include all nonlimited types by definition) have to be externally streamable. 17.e/2

Aspect Description for Pure: Side effects are avoided in the subprograms of a given package. 17.f/3

Erroneous Execution

{*AI12-0076-1*} Execution is erroneous if some operation (other than the initialization or finalization of the object) modifies the value of a constant object declared at library-level in a pure package. 17.1/4

Discussion: This could be accomplished via a self-referencing pointer or via squirrelling a writable pointer to a controlled object. 17.g/4

Implementation Permissions

{*AI95-00366-01*} {*AI05-0219-1*} If a library unit is declared pure, then the implementation is permitted to omit a call on a library-level subprogram of the library unit if the results are not needed after the call. In addition, the implementation may omit a call on such a subprogram and simply reuse the results produced by an earlier call on the same subprogram, provided that none of the parameters nor any object accessible via access values from the parameters have any part that is of a type whose full type is an immutably limited type, and the addresses and values of all by-reference actual parameters, the values of all by-copy-in actual parameters, and the values of all objects accessible via access values from the parameters, are the same as they were at the earlier call. [This permission applies even if the subprogram produces other side effects when called.] 18/3

Discussion: {*AI95-00366-01*} {*AI05-0005-1*} {*AI05-0299-1*} A declared-pure library_item has no variable state. Hence, a call on one of its (nonnested) subprograms cannot normally have side effects. Side effects are still possible via dispatching calls and via indirect calls through access-to-subprogram values. Other mechanisms that might be used to modify variable state include machine code insertions, imported subprograms, and unchecked conversion to an access type declared within the subprogram; this list is not exhaustive. Thus, the permissions described in this subclause may apply to a subprogram whose execution has side effects. The compiler may omit a call to such a subprogram even if side effects exist, so the writer of such a subprogram has to keep this in mind. 18.a/3

Syntax

19/5 {*AI12-0417-1*} The following pragmas are defined with the given forms:

20 **pragma** Elaborate(*library_unit*_name{, *library_unit*_name});

21 **pragma** Elaborate_All(*library_unit*_name{, *library_unit*_name});

22/5 *This paragraph was deleted.*{*AI12-0417-1*}

23 A pragma Elaborate or Elaborate_All is only allowed within a context_clause.

23.a **Ramification:** "Within a context_clause" allows it to be the last item in the context_clause. It can't be first, because the name has to denote something mentioned earlier.

24/5 *This paragraph was deleted.*{*AI12-0417-1*}

24.a **Discussion:** Hence, a pragma Elaborate or Elaborate_All is not elaborated, not that it makes any practical difference.

24.b Note that a pragma Elaborate or Elaborate_All is neither a program unit pragma, nor a library unit pragma.

Legality Rules

25/5 {*AI05-0229-1*} {*AI12-0417-1*} If the aspect Elaborate_Body is True for a declaration, then the declaration requires a completion [(a body)].

25.1/2 {*AI95-00217-06*} The *library_unit*_name of a pragma Elaborate or Elaborate_All shall denote a nonlimited view of a library unit.

25.b/2 **Reason:** These pragmas are intended to prevent elaboration check failures. But a limited view does not make anything visible that has an elaboration check, so the pragmas cannot do anything useful. Moreover, the pragmas would probably reintroduce the circularity that the limited_with_clause was intended to break. So we make such uses illegal.

Static Semantics

26/3 {*AI05-0229-1*} [A pragma Elaborate specifies that the body of the named library unit is elaborated before the current library_item. A pragma Elaborate_All specifies that each library_item that is needed by the named library unit declaration is elaborated before the current library_item.]

26.a **Proof:** The official statement of the semantics of these pragmas is given in 10.2.

26.1/5 {*AI05-0229-1*} {*AI12-0417-1*} [If the Elaborate_Body aspect of a library unit is True, the body of the library unit is elaborated immediately after its declaration.]

26.a.1/3 **Proof:** The official statement of the semantics of this aspect is given in 10.2.

26.b/5 **Implementation Note:** {*AI12-0417-1*} The specification of the Elaborate_Body aspect simplifies the removal of unnecessary Elaboration_Checks. For a subprogram declared immediately within a library unit for which the Elaborate_Body aspect is specified as True, the only calls that can fail the Elaboration_Check are those that occur in the library unit itself, between the declaration and body of the called subprogram; if there are no such calls (which can easily be detected at compile time if there are no stubs), then no Elaboration_Checks are needed for that subprogram. The same is true for Elaboration_Checks on task activations and instantiations, and for library subprograms and generic units.

26.c **Ramification:** The fact that the unit of elaboration is the library_item means that if a subprogram_body is not a completion, it is impossible for any library_item to be elaborated between the declaration and the body of such a subprogram. Therefore, it is impossible for a call to such a subprogram to fail its Elaboration_Check.

26.d **Discussion:** The visibility rules imply that each *library_unit*_name of a pragma Elaborate or Elaborate_All has to denote a library unit mentioned by a previous with_clause of the same context_clause.

26.e/5 **Aspect Description for Elaborate_Body:** A given package will have a body, and that body is elaborated immediately after the declaration.

27/5 NOTE 1 {*AI12-0447-1*} A preelaborated library unit can have nonpreelaborable children.

27.a/1 **Ramification:** {*8652/0035*} {*AI95-00002-01*} But generally not nonpreelaborated subunits. (Nonpreelaborated subunits of subprograms are allowed as discussed above.)

28/5 NOTE 2 {*AI12-0447-1*} A library unit that is declared pure can have impure children.

Ramification: {*8652/0035*} {*AI95-00002-01*} But generally not impure subunits. (Impure subunits of subprograms are allowed as discussed above.) 28.a/1

Ramification: Pragma Elaborate is mainly for closely related library units, such as when two package bodies 'with' each other's declarations. In such cases, Elaborate_All sometimes won't work. 28.b

Extensions to Ada 83

The concepts of preelaborability and purity are new to Ada 95. The Elaborate_All, Elaborate_Body, Preelaborate, and Pure pragmas are new to Ada 95. 28.c

Pragmas Elaborate are allowed to be mixed in with the other things in the context_clause — in Ada 83, they were required to appear last. 28.d

Incompatibilities With Ada 95

{*AI95-00366-01*} The requirement that a partial view with available stream attributes be externally streamable can cause an incompatibility in rare cases. If there is a limited tagged type declared in a pure package with available attributes, and that type is used to declare a private extension in another pure package, and the full type for the private extension has a component of an explicitly limited record type, a protected type, or a type with access discriminants, then the stream attributes will have to be user-specified in the visible part of the package. That is not a requirement for Ada 95, but this combination seems very unlikely in pure packages. Note that this cannot be an incompatibility for a nonlimited type, as all of the types that are allowed in Ada 95 that would require explicitly defined stream attributes are limited (and thus cannot be used as components in a nonlimited type). 28.e/2

{*AI95-00403-01*} **Amendment Correction:** Added wording to cover missing cases for preelaborated generic units. This is incompatible as a preelaborated unit could have used a formal object to initialize a library-level object; that isn't allowed in Ada 2005. But such a unit wouldn't really be preelaborable, and Ada 95 compilers can reject such units (as this is a Binding Interpretation), so such units should be very rare. 28.f/2

Extensions to Ada 95

{*AI95-00161-01*} **Amendment Correction:** The concept of preelaborable initialization and pragma Preelaborable_Initialization are new. These allow more types of objects to be created in preelaborable units, and fix holes in the old rules. 28.g/2

{*AI95-00366-01*} Access-to-subprogram types and access-to-object types with a Storage_Size of 0 are allowed in pure units. The permission to omit calls was adjusted accordingly (which also fixes a hole in Ada 95, as access parameters are allowed, and changes in the values accessed by them must be taken into account). 28.h/2

Wording Changes from Ada 95

{*AI95-00002-01*} **Corrigendum:** The wording was changed so that subunits of a preelaborated subprogram are also preelaborated. 28.i/2

{*AI95-00217-06*} Disallowed pragma Elaborate and Elaborate_All for packages that are mentioned in a limited_with_clause. 28.j/2

Incompatibilities With Ada 2005

{*AI05-0028-1*} **Correction:** Corrected a serious unintended incompatibility with Ada 95 in the new preelaboration wording — explicit initialization of objects of types that don't have preelaborable initialization was not allowed. Ada 2012 switches back to the Ada 95 rule in these cases. This is unlikely to occur in practice, as it is unlikely that a compiler would have implemented the more restrictive rule (it would fail many ACATS tests if it did). 28.k/3

{*AI05-0035-1*} **Correction:** Added an assume-the-worst rule for generic bodies (else they would never be checked for purity) and added the boilerplate so that the entire generic specification is rechecked. Also fixed wording to have consistent handling for subunits for Pure and Preelaborate. An Ada 95 program could have depended on marking a generic pure that was not really pure, although this would defeat the purpose of the categorization and likely cause problems with distributed programs. 28.l/3

Extensions to Ada 2005

{*AI05-0035-1*} **Correction:** Adjusted wording so that a subunit can be pure (it is not a library_item, but it is a compilation unit). 28.m/3

{*AI05-0035-1*} **Correction:** Adjusted wording so that the rules for access types only apply to nonderived types (derived types share their storage pool with their parent, so if the parent access type is legal, so is any derived type.) 28.n/3

{*AI05-0229-1*} {*AI12-0417-1*} Elaborate_Body is now an aspect, so it can be specified by an aspect_specification 28.o/5

28.p/5 {*AI05-0243-1*} {*AI12-0417-1*} Pure and Preelaborate are now aspects, so they can be specified by an aspect_specification

Wording Changes from Ada 2005

28.q/3 {*AI05-0034-1*} **Correction:** Added wording so that a limited view is always treated as pure, no matter what categorization is used for the originating unit. This was undefined in Ada 2005.

28.r/3 {*AI05-0028-1*} {*AI05-0221-1*} **Correction:** Fixed minor issues with preelaborable initialization (PI): null Initialize procedures do not make a type non-PI; formal types with pragma PI can be assumed to have PI; formal extensions are assumed to not have PI; all composite types can have pragma PI (so that the possibility of hidden Initialize routines can be handled); added discriminants of a derived type are not considered in calculating PI.

28.s/3 {*AI05-0219-1*} **Correction:** Clarified that the implementation permission to omit pure subprogram calls does not apply if any part of the parameters or any designated object has a part that is immutably limited. The old wording just said "limited type", which can change via visibility and thus isn't appropriate for dynamic semantics permissions.

Incompatibilities With Ada 2012

28.t/5 {*AI12-0232-1*} **Correction:** Added a rule that a generic declaration is not pure if no instance could be pure (for instance, because the generic declaration would elaborates a variable.) This was legal in Ada 2012, but of course no instance could have been declared pure. A similar rule (with a similar effect) was added for preelaborable generic declarations.

Extensions to Ada 2012

28.u/5 {*AI12-0175-1*} Added some intrinsic conversion functions to those allowed to be called during the elaboration of a preelaborated unit. This is necessary to allow a portable Address aspect in a preelaborated unit, important on small embedded systems.

28.v/5 {*AI12-0409-1*} Aspect Preelaborable_Initialization is new; pragma Preelaborable_Initialization is now obsolescent.

Wording Changes from Ada 2012

28.w/4 {*AI12-0076-1*} **Corrigendum:** Explicitly stated that modifying a library-level constant in a pure package is erroneous. We don't document this as inconsistent as implementations certainly can still do whatever they were previously doing (no change is required); moreover, this case (and many more) were erroneous in Ada 2005 and before, so we're just restoring the previous semantics.

28.x/5 {*AI12-0232-1*} **Correction:** Explicitly stated that the pure and preelaborate rules are recursive; that is, they apply to the contents of nested packages and generic packages.

28.y/5 {*AI12-0417-1*} The pragmas that set aspects (Pure, Preelaborate, Elaborate_Body) are now obsolescent.

11 Exceptions

{*AI05-0299-1*} [This clause defines the facilities for dealing with errors or other exceptional situations that arise during program execution.] An *exception* represents a kind of exceptional situation; an occurrence of such a situation (at run time) is called an *exception occurrence*. [To *raise* an exception is to abandon normal program execution so as to draw attention to the fact that the corresponding situation has arisen. Performing some actions in response to the arising of an exception is called *handling* the exception.] 1/3

To be honest: ...or handling the exception occurrence. 1.a

Ramification: For example, an exception End_Error might represent error situations in which an attempt is made to read beyond end-of-file. During the execution of a partition, there might be numerous occurrences of this exception. 1.b

To be honest: {*AI12-0426-1*} When the meaning is clear from the context, we sometimes use "*occurrence*" as a shorthand for "exception occurrence". 1.c/5

Term entry: exception — kind of exceptional situation 1.d/5

Term entry: exception occurrence — run-time occurrence of an exceptional situation 1.e/5

{*AI05-0043-1*} {*AI05-0258-1*} [An exception_declaration declares a name for an exception. An exception can be raised explicitly (for example, by a raise_statement) or implicitly (for example, by the failure of a language-defined check). When an exception arises, control can be transferred to a user-provided exception_handler at the end of a handled_sequence_of_statements, or it can be propagated to a dynamically enclosing execution.] 2/3

Wording Changes from Ada 83

We are more explicit about the difference between an exception and an occurrence of an exception. This is necessary because we now have a type (Exception_Occurrence) that represents exception occurrences, so the program can manipulate them. Furthermore, we say that when an exception is propagated, it is the same occurrence that is being propagated (as opposed to a new occurrence of the same exception). The same issue applies to a re-raise statement. In order to understand these semantics, we have to make this distinction. 2.a

Wording Changes from Ada 2005

{*AI05-0043-1*} **Correction:** Generalized the introductory description of how an exception can be raised so that it does not appear to cover all of the cases. 2.b/3

11.1 Exception Declarations

An exception_declaration declares a name for an exception. 1

Syntax

{*AI05-0183-1*} exception_declaration ::= defining_identifier_list : **exception** [aspect_specification]; 2/3

Static Semantics

Each single exception_declaration declares a name for a different exception. If a generic unit includes an exception_declaration, the exception_declarations implicitly generated by different instantiations of the generic unit refer to distinct exceptions (but all have the same defining_identifier). The particular exception denoted by an exception name is determined at compilation time and is the same regardless of how many times the exception_declaration is elaborated. 3

Reason: We considered removing this requirement inside generic bodies, because it is an implementation burden for implementations that wish to share code among several instances. In the end, it was decided that it would introduce too much implementation dependence. 3.a

3.b **Ramification:** Hence, if an exception_declaration occurs in a recursive subprogram, the exception name denotes the same exception for all invocations of the recursive subprogram. The reason for this rule is that we allow an exception occurrence to propagate out of its declaration's innermost containing master; if exceptions were created by their declarations like other entities, they would presumably be destroyed upon leaving the master; we would have to do something special to prevent them from propagating to places where they no longer exist.

3.c **Ramification:** Exception identities are unique across all partitions of a program.

4 The *predefined* exceptions are the ones declared in the declaration of package Standard: Constraint_Error, Program_Error, Storage_Error, and Tasking_Error[; one of them is raised when a language-defined check fails.]

4.a **Ramification:** The exceptions declared in the language-defined package IO_Exceptions, for example, are not predefined.

Dynamic Semantics

5 The elaboration of an exception_declaration has no effect.

6/5 {*AI12-0445-1*} The execution of any construct raises Storage_Error if there is insufficient storage for that execution. The amount of storage necessary for the execution of constructs is unspecified.

6.a **Ramification:** Note that any execution whatsoever can raise Storage_Error. This allows much implementation freedom in storage management.

Examples

7 *Examples of user-defined exception declarations:*

8
```
Singular : exception;
Error    : exception;
Overflow, Underflow : exception;
```

Inconsistencies With Ada 83

8.a The exception Numeric_Error is now defined in the Obsolescent features Annex, as a rename of Constraint_Error. All checks that raise Numeric_Error in Ada 83 instead raise Constraint_Error in Ada 95. To increase upward compatibility, we also changed the rules to allow the same exception to be named more than once by a given handler. Thus, "**when** Constraint_Error | Numeric_Error =>" will remain legal in Ada 95, even though Constraint_Error and Numeric_Error now denote the same exception. However, it will not be legal to have separate handlers for Constraint_Error and Numeric_Error. This change is inconsistent in the rare case that an existing program explicitly raises Numeric_Error at a point where there is a handler for Constraint_Error; the exception will now be caught by that handler.

Wording Changes from Ada 83

8.b We explicitly define elaboration for exception_declarations.

Extensions to Ada 2005

8.c/3 {*AI05-0183-1*} An optional aspect_specification can be used in a exception_declaration. This is described in 13.1.1.

11.2 Exception Handlers

1 [The response to one or more exceptions is specified by an exception_handler.]

Syntax

2
```
handled_sequence_of_statements ::=
    sequence_of_statements
  [exception
    exception_handler
    {exception_handler}]
```

3/5 {*AI12-0212-1*} exception_handler ::=
when [choice_parameter_specification:] exception_choice {'|' exception_choice} =>

sequence_of_statements

choice_parameter_specification ::= defining_identifier 4

exception_choice ::= *exception*_name | **others** 5

To be honest: "*Handler*" is an abbreviation for "exception_handler". 5.a

{*AI05-0299-1*} Within this clause, we sometimes abbreviate "exception_choice" to "*choice*". 5.b/3

Legality Rules

{*AI12-0022-1*} An *exception*_name of an exception_choice shall denote an exception. 5.1/4

A choice with an *exception*_name *covers* the named exception. A choice with **others** covers all exceptions not named by previous choices of the same handled_sequence_of_statements. Two choices in different exception_handlers of the same handled_sequence_of_statements shall not cover the same exception. 6

Ramification: Two exception_choices of the same exception_handler may cover the same exception. For example, given two renaming declarations in separate packages for the same exception, one may nevertheless write, for example, "**when** Ada.Text_IO.Data_Error | My_Seq_IO.Data_Error =>". 6.a

An **others** choice even covers exceptions that are not visible at the place of the handler. Since exception raising is a dynamic activity, it is entirely possible for an **others** handler to handle an exception that it could not have named. 6.b

A choice with **others** is allowed only for the last handler of a handled_sequence_of_statements and as the only choice of that handler. 7

An *exception*_name of a choice shall not denote an exception declared in a generic formal package. 8

Reason: This is because the compiler doesn't know the identity of such an exception, and thus can't enforce the coverage rules. 8.a

Static Semantics

A choice_parameter_specification declares a *choice parameter*, which is a constant object of type Exception_Occurrence (see 11.4.1). During the handling of an exception occurrence, the choice parameter, if any, of the handler represents the exception occurrence that is being handled. 9

Dynamic Semantics

The execution of a handled_sequence_of_statements consists of the execution of the sequence_of_statements. [The optional handlers are used to handle any exceptions that are propagated by the sequence_of_statements.] 10

Examples

Example of an exception handler: 11

```
{AI12-0178-1} begin                                                    12/5
   Open(File, In_File, "input.txt");    -- see A.8.2
exception
   when E : Name_Error =>
      Put("Cannot open input file : ");
      Put_Line(Ada.Exceptions.Exception_Message(E));  -- see 11.4.1
      raise;
end;
```

Extensions to Ada 83

The syntax rule for exception_handler is modified to allow a choice_parameter_specification. 12.a

{*AI95-00114-01*} Different exception_choices of the same exception_handler may cover the same exception. This allows for "when Numeric_Error | Constraint_Error =>" even though Numeric_Error is a rename of Constraint_Error. This also allows one to "with" two different I/O packages, and then write, for example, "when 12.b/2

Ada.Text_IO.Data_Error | My_Seq_IO.Data_Error =>" even though these might both be renames of the same exception.

Wording Changes from Ada 83

12.c The syntax rule for handled_sequence_of_statements is new. These are now used in all the places where handlers are allowed. This obviates the need to explain (in Clauses 5, 6, 7, and 9) what portions of the program are handled by the handlers. Note that there are more such cases in Ada 95.

12.d The syntax rule for choice_parameter_specification is new.

11.3 Raise Statements and Raise Expressions

1 [A raise_statement raises an exception.]

Syntax

2/2 {*AI95-00361-01*} raise_statement ::= **raise**;
| **raise** *exception_*name [**with** *string_*expression];

2.1/4 {*AI12-0022-1*} {*AI12-0152-1*} raise_expression ::= **raise** *exception_*name [**with** *string_*simple_expression]

2.2/4 {*AI12-0152-1*} If a raise_expression appears within the expression of one of the following contexts, the raise_expression shall appear within a pair of parentheses within the expression:

2.3/4 - object_declaration;
2.4/4 - modular_type_definition;
2.5/4 - floating_point_definition;
2.6/4 - ordinary_fixed_point_definition;
2.7/4 - decimal_fixed_point_definition;
2.8/4 - default_expression;
2.9/4 - ancestor_part.

2.a.1/4 **Reason:** Unlike conditional expressions, this doesn't say "immediately surrounded"; the only requirement is that it is somehow within a pair of parentheses that is part of the expression. We need this restriction in order that raise_expressions cannot be syntactically confused with immediately following constructs (such as aspect_specifications).

2.a.2/4 **Discussion:** We only need to require that a right parenthesis appear somewhere between the raise_expression and the surrounding context; that's all we need to specify in order to eliminate the ambiguities. Moreover, we don't care at all where the left parenthesis is (so long as it is legal, of course).

2.a.3/4 For instance, the following is illegal by this rule:

2.a.4/4
```
Obj : Boolean := Func_Call or else raise TBD_Error with Atomic;
```

2.a.5/4 as the "**with** Atomic" could be part of the **raise_expression** or part of the object declaration. Both of the following are legal:

2.a.6/4
```
Obj : Boolean := Func_Call or else (raise TBD_Error) with Atomic;
Obj : Boolean := (Func_Call or else raise TBD_Error) with Atomic;
```

2.a.7/4 and if the **with** belongs to the raise_expression, then both of the following are legal:

2.a.8/4
```
Obj : Boolean := Func_Call or else (raise TBD_Error with Atomic);
Obj : Boolean := (Func_Call or else raise TBD_Error with Atomic);
```

2.a.9/4 This rule only requires parentheses for raise_expressions that are part of the "top-level" of an expression in one of the named contexts; the raise_expression is either the entire expression, or part of a chain of logical operations. In practice, the raise_expression will almost always be last in interesting top-level expressions; anything that follows it could never be executed, so that should be rare. Other contexts such as conditional expressions, qualified expressions, aggregates, and even function calls, provide the needed parentheses. All of the following are legal, no additional parens are needed:

```
Pre : Boolean  := (if not Is_Valid(Param) then raise Not_Valid_Error);
A : A_Tagged   := (Some_Tagged'(raise TBD_Error) with Comp => 'A');
B : Some_Array := (1, 2, 3, others => raise Not_Valid_Error);
C : Natural    := Func (Val => raise TBD_Error);
```

2.a.10/4

Parentheses that are part of the context of the expression don't count. For instance, the parentheses around the raise_expression are required in the following: 2.a.11/4

```
D : A_Tagged   := ((raise TBD_Error) with Comp => 'A');
```

2.a.12/4

as ancestor_part is one of the contexts that triggers the rule. 2.a.13/4

This English-language rule could have been implemented instead by adding nonterminals initial_expression and initial_relation, which are the same as choice_expression and choice_relation except for the inclusion of membership in initial_relation. Then, initial_expresion could be used in place of expression in all of the contexts noted. We did not do that because of the large amount of change required, both to the grammar and to language rules that refer to the grammar. A complete grammar is given in AI12-0152-1. 2.a.14/4

The use of a raise_expression is illegal in each of modular_type_definition, floating_point_definition, ordinary_fixed_point_definition, and decimal_fixed_point_definition as these uses are required to be static and a raise_expression is never static. We include these in this rule so that Ada text has an unambiguous syntax in these cases. 2.a.15/4

Legality Rules

{*AI12-0022-1*} {*AI12-0159-1*} The *exception*_name, if any, of a raise_statement or raise_expression shall denote an exception. A raise_statement with no *exception*_name (that is, a *re-raise statement*) shall be within a handler, but not within a body enclosed by that handler. 3/4

Name Resolution Rules

{*AI95-00361-01*} {*AI12-0022-1*} {*AI12-0152-1*} The *string*_expression or *string*_simple_expression, if any, of a raise_statement or raise_expression is expected to be of type String. 3.1/4

{*AI12-0022-1*} {*AI12-0159-1*} The expected type for a raise_expression shall be any single type. 3.2/4

Dynamic Semantics

{*AI95-00361-01*} {*AI12-0022-1*} {*AI12-0152-1*} To *raise an exception* is to raise a new occurrence of that exception[, as explained in 11.4]. For the execution of a raise_statement with an *exception*_name, the named exception is raised. Similarly, for the evaluation of a raise_expression, the named exception is raised. [In both of these cases, if a *string*_expression or *string*_simple_expression is present, the expression is evaluated and its value is associated with the exception occurrence.] For the execution of a re-raise statement, the exception occurrence that caused transfer of control to the innermost enclosing handler is raised [again]. 4/4

Proof: {*AI95-00361-01*} The definition of Exceptions.Exception_Message includes a statement that the string is returned (see 11.4.1). We describe the use of the string here so that we don't have an unexplained parameter in this subclause. 4.a.1/2

Implementation Note: For a re-raise statement, the implementation does not create a new Exception_Occurrence, but instead propagates the same Exception_Occurrence value. This allows the original cause of the exception to be determined. 4.a

Term entry: raise an exception — abandon normal program execution so as to draw attention to the fact that the corresponding situation has arisen 4.b/5

NOTE {*AI12-0062-1*} {*AI12-0152-1*} {*AI12-0159-1*} If the evaluation of a *string*_expression or *string*_simple_expression raises an exception, that exception is propagated instead of the one denoted by the *exception*_name of the raise_statement or raise_expression. 4.1/4

Examples

Examples of raise statements: 5

```
{AI95-00433-01} raise Ada.IO_Exceptions.Name_Error;    -- see A.13
raise Queue_Error with "Buffer Full"; -- see 9.11
```

6/2

```
raise;                                 -- re-raise the current exception
```

7

8/5 {AI12-0312-1} -- *For an example of a raise expression, see the Streams Subsystem definitions in 13.13.1.*

Wording Changes from Ada 83

8.a The fact that the name in a raise_statement has to denote an exception is not clear from RM83. Clearly that was the intent, since the italicized part of the syntax rules so indicate, but there was no explicit rule. RM83-1.5(11) doesn't seem to give the italicized parts of the syntax any force.

Extensions to Ada 95

8.b/2 {*AI95-00361-01*} The syntax of a raise_statement is extended to include a string message. This is more convenient than calling Exceptions.Exception_Message (*exception*_name'Identity, *string*_expression), and should encourage the use of message strings when raising exceptions.

Extensions to Ada 2012

8.c/4 {*AI12-0022-1*} {*AI12-0152-1*} {*AI12-0159-1*} **Corrigendum:** The raise_expression is new. This construct is necessary to allow conversion of existing specifications to use preconditions and predicates without changing the exceptions raised. It is considered important enough to be added to Ada 2012 rather than waiting for Ada 2022.

11.4 Exception Handling

1 [When an exception occurrence is raised, normal program execution is abandoned and control is transferred to an applicable exception_handler, if any. To *handle* an exception occurrence is to respond to the exceptional event. To *propagate* an exception occurrence is to raise it again in another context; that is, to fail to respond to the exceptional event in the present context.]

1.a **Ramification:** In other words, if the execution of a given construct raises an exception, but does not handle it, the exception is propagated to an enclosing execution (except in the case of a task_body).

1.b/1 Propagation involves re-raising the same exception occurrence. For example, calling an entry of an uncallable task raises Tasking_Error; this is not propagation.

Dynamic Semantics

2 Within a given task, if the execution of construct *a* is defined by this document to consist (in part) of the execution of construct *b*, then while *b* is executing, the execution of *a* is said to *dynamically enclose* the execution of *b*. The *innermost dynamically enclosing* execution of a given execution is the dynamically enclosing execution that started most recently.

2.a **To be honest:** If the execution of *a* dynamically encloses that of *b*, then we also say that the execution of *b* is *included in* the execution of *a*.

2.b **Ramification:** Examples: The execution of an if_statement dynamically encloses the evaluation of the condition after the **if** (during that evaluation). (Recall that "execution" includes both "elaboration" and "evaluation", as well as other executions.) The evaluation of a function call dynamically encloses the execution of the sequence_of_statements of the function body (during that execution). Note that, due to recursion, several simultaneous executions of the same construct can be occurring at once during the execution of a particular task.

2.c Dynamically enclosing is not defined across task boundaries; a task's execution does not include the execution of any other tasks.

2.d Dynamically enclosing is only defined for executions that are occurring at a given moment in time; if an if_statement is currently executing the sequence_of_statements after **then**, then the evaluation of the condition is no longer dynamically enclosed by the execution of the if_statement (or anything else).

3 When an exception occurrence is raised by the execution of a given construct, the rest of the execution of that construct is *abandoned*; that is, any portions of the execution that have not yet taken place are not performed. The construct is first completed, and then left, as explained in 7.6.1. Then:

4 • If the construct is a task_body, the exception does not propagate further;

4.a **Ramification:** When an exception is raised by the execution of a task_body, there is no dynamically enclosing execution, so the exception does not propagate any further. If the exception occurred during the activation of the task, then the activator raises Tasking_Error, as explained in 9.2, "Task Execution - Task Activation", but we don't define that as propagation; it's a special rule. Otherwise (the exception occurred during the execution of the

handled_sequence_of_statements of the task), the task silently disappears. Thus, abnormal termination of tasks is not always considered to be an error.

- If the construct is the sequence_of_statements of a handled_sequence_of_statements that has a handler with a choice covering the exception, the occurrence is handled by that handler; 5
- Otherwise, the occurrence is *propagated* to the innermost dynamically enclosing execution, which means that the occurrence is raised again in that context. 6

To be honest: As shorthands, we refer to the *propagation of an exception*, and the *propagation by a construct*, if the execution of the construct propagates an exception occurrence. 6.a

When an occurrence is *handled* by a given handler, the choice_parameter_specification, if any, is first elaborated, which creates the choice parameter and initializes it to the occurrence. Then, the sequence_of_statements of the handler is executed; this execution replaces the abandoned portion of the execution of the sequence_of_statements. 7

Ramification: {*AI95-00318-02*} This "replacement" semantics implies that the handler can do pretty much anything the abandoned sequence could do; for example, in a function, the handler can execute a return statement that applies to the function. 7.a/2

Ramification: The rules for exceptions raised in library units, main subprograms and partitions follow from the normal rules, plus the semantics of the environment task described in Clause 10 (for example, the environment task of a partition elaborates library units and calls the main subprogram). If an exception is propagated by the main subprogram, it is propagated to the environment task, which then terminates abnormally, causing the partition to terminate abnormally. Although abnormal termination of tasks is not necessarily an error, abnormal termination of a partition due to an exception *is* an error. 7.b

Term entry: handle an exception — perform some actions in response to the arising of an exception 7.c/5

NOTE {*AI12-0451-1*} Exceptions raised in a declarative_part of a body are not handled by the handlers of the handled_sequence_of_statements of that body. 8/5

11.4.1 The Package Exceptions

Static Semantics

The following language-defined library package exists: 1

```
{AI95-00362-01} {AI95-00400-01} {AI95-00438-01} {AI12-0241-1} {AI12-0302-1} {AI12-0399-1} with Ada.Streams;
package Ada.Exceptions
    with Preelaborate, Nonblocking, Global => in out synchronized is
    type Exception_Id is private
        with Preelaborable_Initialization;
    Null_Id : constant Exception_Id;
    function Exception_Name(Id : Exception_Id) return String;
    function Wide_Exception_Name(Id : Exception_Id) return Wide_String;
    function Wide_Wide_Exception_Name(Id : Exception_Id)
        return Wide_Wide_String;
```

2/5

```
{AI95-00362-01} {AI12-0399-1}     type Exception_Occurrence is limited private
        with Preelaborable_Initialization;
    type Exception_Occurrence_Access is access all Exception_Occurrence;
    Null_Occurrence : constant Exception_Occurrence;
```

3/5

```
{AI95-00329-01} {AI05-0229-1}     procedure Raise_Exception(E : in Exception_Id;
                                      Message : in String := "")
        with No_Return;
    function Exception_Message(X : Exception_Occurrence) return String;
    procedure Reraise_Occurrence(X : in Exception_Occurrence);
```

4/3

5/2 {*AI95-00400-01*}
```
function Exception_Identity(X : Exception_Occurrence)
                            return Exception_Id;
function Exception_Name(X : Exception_Occurrence) return String;
   -- Same as Exception_Name(Exception_Identity(X)).
function Wide_Exception_Name(X : Exception_Occurrence)
   return Wide_String;
   -- Same as Wide_Exception_Name(Exception_Identity(X)).
function Wide_Wide_Exception_Name(X : Exception_Occurrence)
   return Wide_Wide_String;
   -- Same as Wide_Wide_Exception_Name(Exception_Identity(X)).
function Exception_Information(X : Exception_Occurrence) return String;
```

6/2 {*AI95-00438-01*}
```
procedure Save_Occurrence(Target : out Exception_Occurrence;
                          Source : in Exception_Occurrence);
function Save_Occurrence(Source : Exception_Occurrence)
                         return Exception_Occurrence_Access;
```

6.1/2 {*AI95-00438-01*}
```
procedure Read_Exception_Occurrence
   (Stream : not null access Ada.Streams.Root_Stream_Type'Class;
    Item   : out Exception_Occurrence);
procedure Write_Exception_Occurrence
   (Stream : not null access Ada.Streams.Root_Stream_Type'Class;
    Item   : in Exception_Occurrence);
```

6.2/2 {*AI95-00438-01*}
```
for Exception_Occurrence'Read use Read_Exception_Occurrence;
for Exception_Occurrence'Write use Write_Exception_Occurrence;
```

6.3/2 {*AI95-00438-01*}
```
private
   ... -- not specified by the language
end Ada.Exceptions;
```

7 Each distinct exception is represented by a distinct value of type Exception_Id. Null_Id does not represent any exception, and is the default initial value of type Exception_Id. Each occurrence of an exception is represented by a value of type Exception_Occurrence. Null_Occurrence does not represent any exception occurrence, and is the default initial value of type Exception_Occurrence.

8/1 For a prefix E that denotes an exception, the following attribute is defined:

9 E'Identity E'Identity returns the unique identity of the exception. The type of this attribute is Exception_Id.

9.a **Ramification:** In a distributed program, the identity is unique across an entire program, not just across a single partition. Exception propagation works properly across RPC's. An exception can be propagated from one partition to another, and then back to the first, where its identity is known.

10/2 {*AI95-00361-01*} Raise_Exception raises a new occurrence of the identified exception.

10.1/4 {*AI95-00361-01*} {*AI95-00378-01*} {*AI05-0043-1*} {*AI05-0248-1*} {*AI12-0022-1*} {*AI12-0152-1*} Exception_Message returns the message associated with the given Exception_Occurrence. For an occurrence raised by a call to Raise_Exception, the message is the Message parameter passed to Raise_Exception. For the occurrence raised by a raise_statement or raise_expression with an *exception*_name and a *string*_expression or *string*_simple_expression, the message is the *string*_expression or *string*_simple_expression. For the occurrence raised by a raise_statement or raise_expression with an *exception*_name but without a *string*_expression or *string*_simple_expression, the message is a string giving implementation-defined information about the exception occurrence. For an occurrence originally raised in some other manner (including by the failure of a language-defined check), the message is an unspecified string. In all cases, Exception_Message returns a string with lower bound 1.

10.a **Implementation defined:** The information returned by Exception_Message.

10.a.1/3 **Discussion:** {*AI05-0043-1*} There is Implementation Advice about the contents of this string for language-defined checks.

10.b **Ramification:** Given an exception E, the raise_statement:

10.c
```
raise E;
```

is equivalent to this call to Raise_Exception: 10.d

```
Raise_Exception(E'Identity, Message => implementation-defined-string);
```
10.e

{*AI95-00361-01*} Similarly, the raise_statement: 10.e.1/2

```
raise E with "some information";
```
10.e.2/2

is equivalent to this call to Raise_Exception: 10.e.3/2

```
Raise_Exception(E'Identity, Message => "some information");
```
10.e.4/2

{*AI95-00361-01*} Reraise_Occurrence reraises the specified exception occurrence. 10.2/2

Ramification: The following handler: 10.f

```
when others =>
    Cleanup;
    raise;
```
10.g

is equivalent to this one: 10.h

```
when X : others =>
    Cleanup;
    Reraise_Occurrence(X);
```
10.i

Exception_Identity returns the identity of the exception of the occurrence. 11

{*AI95-00400-01*} The Wide_Wide_Exception_Name functions return the full expanded name of the exception, in upper case, starting with a root library unit. For an exception declared immediately within package Standard, the defining_identifier is returned. The result is implementation defined if the exception is declared within an unnamed block_statement. 12/2

Ramification: See the Implementation Permission below. 12.a

To be honest: This name, as well as each prefix of it, does not denote a renaming_declaration. 12.b

Implementation defined: The result of Exceptions.Wide_Wide_Exception_Name for exceptions declared within an unnamed block_statement. 12.c/2

Ramification: Note that we're talking about the name of the exception, not the name of the occurrence. 12.d

{*AI95-00400-01*} The Exception_Name functions (respectively, Wide_Exception_Name) return the same sequence of graphic characters as that defined for Wide_Wide_Exception_Name, if all the graphic characters are defined in Character (respectively, Wide_Character); otherwise, the sequence of characters is implementation defined, but no shorter than that returned by Wide_Wide_Exception_Name for the same value of the argument. 12.1/2

Implementation defined: The sequence of characters of the value returned by Exceptions.Exception_Name (respectively, Exceptions.Wide_Exception_Name) when some of the graphic characters of Exceptions.Wide_Wide_Exception_Name are not defined in Character (respectively, Wide_Character). 12.e/2

{*AI95-00378-01*} {*AI95-00417-01*} The string returned by the Exception_Name, Wide_Exception_Name, and Wide_Wide_Exception_Name functions has lower bound 1. 12.2/2

{*AI95-00378-01*} Exception_Information returns implementation-defined information about the exception occurrence. The returned string has lower bound 1. 13/2

Implementation defined: The information returned by Exception_Information. 13.a

{*AI95-00241-01*} {*AI95-00446-01*} Reraise_Occurrence has no effect in the case of Null_Occurrence. Raise_Exception and Exception_Name raise Constraint_Error for a Null_Id. Exception_Message, Exception_Name, and Exception_Information raise Constraint_Error for a Null_Occurrence. Exception_Identity applied to Null_Occurrence returns Null_Id. 14/2

Ramification: {*AI95-00241-01*} Null_Occurrence can be tested for by comparing Exception_Identity(Occurrence) to Null_Id. 14.a.1/2

Discussion: {*AI95-00446-01*} Raise_Exception was changed so that it always raises an exception and thus can be a No_Return procedure. A similar change was not made for Reraise_Occurrence, as doing so was determined to be a significant incompatibility. It is not unusual to pass an Exception_Occurrence to other code to delay raising it. If there was no exception, passing Null_Occurrence works fine (nothing is raised). Moreover, as there is no test for 14.a.2/2

Null_Occurrence in Ada 95, this is the only way to write such code without using additional flags. Breaking this sort of code is unacceptable.

15 The Save_Occurrence procedure copies the Source to the Target. The Save_Occurrence function uses an allocator of type Exception_Occurrence_Access to create a new object, copies the Source to this new object, and returns an access value designating this new object; [the result may be deallocated using an instance of Unchecked_Deallocation.]

15.a **Ramification:** It's OK to pass Null_Occurrence to the Save_Occurrence subprograms; they don't raise an exception, but simply save the Null_Occurrence.

15.1/2 {*AI95-00438-01*} Write_Exception_Occurrence writes a representation of an exception occurrence to a stream; Read_Exception_Occurrence reconstructs an exception occurrence from a stream (including one written in a different partition).

15.b/5 **Ramification:** {*AI12-0005-1*} These routines are used to define the stream attributes (see 13.13.2) for Exception_Occurrence.

15.c/2 The identity of the exception, as well as the Exception_Name and Exception_Message, have to be preserved across partitions.

15.d/2 The string returned by Exception_Name or Exception_Message on the result of calling the Read attribute on a given stream has to be the same as the value returned by calling the corresponding function on the exception occurrence that was written into the stream with the Write attribute. The string returned by Exception_Information need not be the same, since it is implementation defined anyway.

15.e/2 **Reason:** This is important for supporting writing exception occurrences to external files for post-mortem analysis, as well as propagating exceptions across remote subprogram calls in a distributed system (see E.4).

Paragraph 16 was deleted.

Implementation Permissions

17 An implementation of Exception_Name in a space-constrained environment may return the defining_identifier instead of the full expanded name.

18 The string returned by Exception_Message may be truncated (to no less than 200 characters) by the Save_Occurrence procedure [(not the function)], the Reraise_Occurrence procedure, and the re-raise statement.

18.a **Reason:** The reason for allowing truncation is to ease implementations. The reason for choosing the number 200 is that this is the minimum source line length that implementations have to support, and this feature seems vaguely related since it's usually a “one-liner”. Note that an implementation is allowed to do this truncation even if it supports arbitrarily long lines.

Implementation Advice

19 Exception_Message (by default) and Exception_Information should produce information useful for debugging. Exception_Message should be short (about one line), whereas Exception_Information can be long. Exception_Message should not include the Exception_Name. Exception_Information should include both the Exception_Name and the Exception_Message.

19.a.1/2 **Implementation Advice:** Exception_Information should provide information useful for debugging, and should include the Exception_Name and Exception_Message.

19.a.2/2 **Implementation Advice:** Exception_Message by default should be short, provide information useful for debugging, and should not include the Exception_Name.

19.a **Reason:** It may seem strange to define two subprograms whose semantics is implementation defined. The idea is that a program can print out debugging/error-logging information in a portable way. The program is portable in the sense that it will work in any implementation; it might print out different information, but the presumption is that the information printed out is appropriate for debugging/error analysis on that system.

19.b **Implementation Note:** As an example, Exception_Information might include information identifying the location where the exception occurred, and, for predefined exceptions, the specific kind of language-defined check that failed. There is an implementation trade-off here, between how much information is represented in an Exception_Occurrence, and how much can be passed through a re-raise.

The string returned should be in a form suitable for printing to an error log file. This means that it might need to contain line-termination control characters with implementation-defined I/O semantics. The string should neither start nor end with a newline. 19.c

If an implementation chooses to provide additional functionality related to exceptions and their occurrences, it should do so by providing one or more children of Ada.Exceptions. 19.d

Note that exceptions behave as if declared at library level; there is no “natural scope” for an exception; an exception always exists. Hence, there is no harm in saving an exception occurrence in a data structure, and reraising it later. The reraise has to occur as part of the same program execution, so saving an exception occurrence in a file, reading it back in from a different program execution, and then reraising it is not required to work. This is similar to I/O of access types. Note that it is possible to use RPC to propagate exceptions across partitions. 19.e

Here's one way to implement Exception_Occurrence in the private part of the package. Using this method, an implementation need store only the actual number of characters in exception messages. If the user always uses small messages, then exception occurrences can be small. If the user never uses messages, then exception occurrences can be smaller still: 19.f

```
type Exception_Occurrence(Message_Length : Natural := 200) is
    limited record
        Id : Exception_Id;
        Message : String(1..Message_Length);
    end record;
```
19.g

At the point where an exception is raised, an Exception_Occurrence can be allocated on the stack with exactly the right amount of space for the message — none for an empty message. This is just like declaring a constrained object of the type: 19.h

```
Temp : Exception_Occurrence(10); -- for a 10-character message
```
19.i

After finding the appropriate handler, the stack can be cut back, and the Temp copied to the right place. This is similar to returning an unknown-sized object from a function. It is not necessary to allocate the maximum possible size for every Exception_Occurrence. If, however, the user declares an Exception_Occurrence object, the discriminant will be permanently set to 200. The Save_Occurrence procedure would then truncate the Exception_Message. Thus, nothing is lost until the user tries to save the occurrence. If the user is willing to pay the cost of heap allocation, the Save_Occurrence function can be used instead. 19.j

Note that any arbitrary-sized implementation-defined Exception_Information can be handled in a similar way. For example, if the Exception_Occurrence includes a stack traceback, a discriminant can control the number of stack frames stored. The traceback would be truncated or entirely deleted by the Save_Occurrence procedure — as the implementation sees fit. 19.k

If the internal representation involves pointers to data structures that might disappear, it would behoove the implementation to implement it as a controlled type, so that assignment can either copy the data structures or else null out the pointers. Alternatively, if the data structures being pointed at are in a task control block, the implementation could keep a unique sequence number for each task, so it could tell when a task's data structures no longer exist. 19.l

Using the above method, heap space is never allocated unless the user calls the Save_Occurrence function. 19.m

An alternative implementation would be to store the message strings on the heap when the exception is raised. (It could be the global heap, or it could be a special heap just for this purpose — it doesn't matter.) This representation would be used only for choice parameters. For normal user-defined exception occurrences, the Save_Occurrence procedure would copy the message string into the occurrence itself, truncating as necessary. Thus, in this implementation, Exception_Occurrence would be implemented as a variant record: 19.n

```
type Exception_Occurrence_Kind is (Normal, As_Choice_Param);
```
19.o

```
type Exception_Occurrence(Kind : Exception_Occurrence_Kind := Normal) is
    limited record
        case Kind is
            when Normal =>
                ... -- space for 200 characters
            when As_Choice_Param =>
                ... -- pointer to heap string
        end case;
    end record;
```
19.p

Exception_Occurrences created by the run-time system during exception raising would be As_Choice_Param. User-declared ones would be Normal — the user cannot see the discriminant, and so cannot set it to As_Choice_Param. The strings in the heap would be freed upon completion of the handler. 19.q

This alternative implementation corresponds to a heap-based implementation of functions returning unknown-sized results. 19.r

One possible implementation of Reraise_Occurrence is as follows: 19.s

19.t
```
procedure Reraise_Occurrence(X : in Exception_Occurrence) is
begin
    Raise_Exception(Identity(X), Exception_Message(X));
end Reraise_Occurrence;
```

19.u However, some implementations may wish to retain more information across a re-raise — a stack traceback, for example.

19.v **Ramification:** Note that Exception_Occurrence is a definite subtype. Hence, values of type Exception_Occurrence may be written to an error log for later analysis, or may be passed to subprograms for immediate error analysis.

19.w/2 *This paragraph was deleted.*{*AI95-00400-01*}

20/5 NOTE {*AI12-0021-1*} UTF-8 encoding (see A.4.11) can be used to represent non-ASCII characters in exception messages.

Extensions to Ada 83

20.a The Identity attribute of exceptions is new, as is the package Exceptions.

Inconsistencies With Ada 95

20.b/2 {*AI95-00241-01*} **Amendment Correction:** Exception_Identity of an Exception_Occurrence now is defined to return Null_Id for Null_Occurrence, rather than raising Constraint_Error. This provides a simple way to test for Null_Occurrence. We expect that programs that need Constraint_Error raised will be very rare; they can be easily fixed by explicitly testing for Null_Id or by using Exception_Name instead.

20.c/2 {*AI95-00378-01*} {*AI95-00417-01*} **Amendment Correction:** We now define the lower bound of the string returned from [[Wide_]Wide_]Exception_Name, Exception_Message, and Exception_Information. This makes working with the returned string easier, and is consistent with many other string-returning functions in Ada. This is technically an inconsistency; if a program depended on some other lower bound for the string returned from one of these functions, it could fail when compiled with Ada 2005. Such code is not portable even between Ada 95 implementations, so it should be very rare.

20.d/2 {*AI95-00446-01*} **Amendment Correction:** Raise_Exception now raises Constraint_Error if passed Null_Id. This means that it always raises an exception, and thus we can apply pragma No_Return to it. We expect that programs that call Raise_Exception with Null_Id will be rare, and programs that do that and expect no exception to be raised will be rarer; such programs can be easily fixed by explicitly testing for Null_Id before calling Raise_Exception.

Incompatibilities With Ada 95

20.e/3 {*AI95-00400-01*} {*AI95-00438-01*} {*AI05-0005-1*} Functions Wide_Exception_Name and Wide_Wide_Exception_Name, and procedures Read_Exception_Occurrence and Write_Exception_Occurrence are added to Exceptions. If Exceptions is referenced in a use_clause, and an entity *E* with the same defining_identifier as a new entity in Exceptions is defined in a package that is also referenced in a use_clause, the entity *E* may no longer be use-visible, resulting in errors. This should be rare and is easily fixed if it does occur.

Extensions to Ada 95

20.f/2 {*AI95-00362-01*} The package Exceptions is preelaborated, and types Exception_Id and Exception_Occurrence have preelaborable initialization, allowing this package to be used in preelaborated units.

Wording Changes from Ada 95

20.g/2 {*AI95-00361-01*} The meaning of Exception_Message is reworded to reflect that the string can come from a raise_statement as well as a call of Raise_Exception.

20.h/2 {*AI95-00400-01*} Added Wide_Exception_Name and Wide_Wide_Exception_Name because identifiers can now contain characters outside of Latin-1.

Wording Changes from Ada 2005

20.i/3 {*AI05-0043-1*} **Correction:** Added explicit wording that the exception message for language-defined checks is unspecified. The old wording appeared inclusive, but it was not.

11.4.2 Pragmas Assert and Assertion_Policy

1/3 {*AI95-00286-01*} {*AI05-0274-1*} Pragma Assert is used to assert the truth of a boolean expression at a point within a sequence of declarations or statements.

{*AI05-0274-1*} {*AI12-0265-1*} Assert pragmas, subtype predicates (see 3.2.4), preconditions and postconditions (see 6.1.1), type invariants (see 7.3.2), and default initial conditions (see 7.3.3) are collectively referred to as *assertions*; their boolean expressions are referred to as *assertion expressions*. 1.1/5

Term entry: assertion — boolean expression that is expected to be True at run time at certain specified places 1.a/5
Note: Certain pragmas and aspects define various kinds of assertions.

{*AI05-0274-1*} Pragma Assertion_Policy is used to control whether assertions are to be ignored by the implementation, checked at run time, or handled in some implementation-defined manner. 1.2/3

Syntax

{*AI95-00286-01*} The form of a pragma Assert is as follows: 2/2

pragma Assert([Check =>] *boolean*_expression[, [Message =>] *string*_expression]); 3/2

A pragma Assert is allowed at the place where a declarative_item or a statement is allowed. 4/2

{*AI95-00286-01*} The form of a pragma Assertion_Policy is as follows: 5/2

pragma Assertion_Policy(*policy*_identifier); 6/2

{*AI05-0290-1*} **pragma** Assertion_Policy(6.1/3
*assertion*_aspect_mark => *policy*_identifier
{, *assertion*_aspect_mark => *policy*_identifier});

{*AI05-0290-1*} A pragma Assertion_Policy is allowed only immediately within a declarative_part, immediately within a package_specification, or as a configuration pragma. 7/3

Name Resolution Rules

{*AI95-00286-01*} The expected type for the *boolean*_expression of a pragma Assert is any boolean type. The expected type for the *string*_expression of a pragma Assert is type String. 8/2

Reason: We allow any boolean type to be like if_statements and other conditionals; we only allow String for the message in order to match raise_statements. 8.a/2

Legality Rules

{*AI95-00286-01*} {*AI05-0290-1*} {*AI12-0265-1*} {*AI12-0396-1*} The *assertion*_aspect_mark of a pragma Assertion_Policy shall identify an *assertion aspect*, namely one of Assert, Static_Predicate, Dynamic_Predicate, Pre, Pre'Class, Post, Post'Class, Type_Invariant, Type_Invariant'Class, Default_Initial_Condition, or some implementation-defined (assertion) aspect_mark. The *policy*_identifier shall be either Check, Ignore, or some implementation-defined identifier. 9/5

Implementation defined: Implementation-defined *policy*_identifiers and *assertion*_aspect_marks allowed in a pragma Assertion_Policy. 9.a/3

To be honest: {*AI12-0396-1*} "Assert" is considered an "assertion aspect" for the purposes of this rule, even though there is no sort of entity that has an Assert aspect. It can only be specified using an Assert pragma, and applies to a particular point in the execution of a logical thread of control. 9.b/5

Static Semantics

{*AI95-00286-01*} {*AI05-0290-1*} A pragma Assertion_Policy determines for each assertion aspect named in the pragma_argument_associations whether assertions of the given aspect are to be enforced by a runtime check. The *policy*_identifier Check requires that assertion expressions of the given aspect be checked that they evaluate to True at the points specified for the given aspect; the *policy*_identifier Ignore requires that the assertion expression not be evaluated at these points, and the runtime checks not be performed. [Note that for subtype predicate aspects (see 3.2.4), even when the applicable Assertion_Policy is Ignore, the predicate will still be evaluated as part of membership tests and Valid 10/3

attribute_references, and if static, will still have an effect on loop iteration over the subtype, and the selection of case_statement_alternatives and variants.]

10.1/3 {*AI05-0290-1*} If no *assertion_*aspect_marks are specified in the pragma, the specified policy applies to all assertion aspects.

10.2/5 {*AI05-0290-1*} {*AI12-0396-1*} A pragma Assertion_Policy applies to the named assertion aspects in a specific region, and applies to all assertion expressions associated with those aspects specified in that region. A pragma Assertion_Policy given in a declarative_part or immediately within a package_specification applies from the place of the pragma to the end of the innermost enclosing declarative region. The region for a pragma Assertion_Policy given as a configuration pragma is the declarative region for the entire compilation unit (or units) to which it applies.

10.3/3 {*AI05-0290-1*} If a pragma Assertion_Policy applies to a generic_instantiation, then the pragma Assertion_Policy applies to the entire instance.

10.a.1/3 **Ramification:** This means that an Assertion_Policy pragma that occurs in a scope enclosing the declaration of a generic unit but not also enclosing the declaration of a given instance of that generic unit will not apply to assertion expressions occurring within the given instance.

10.4/3 {*AI05-0290-1*} If multiple Assertion_Policy pragmas apply to a given construct for a given assertion aspect, the assertion policy is determined by the one in the innermost enclosing region of a pragma Assertion_Policy specifying a policy for the assertion aspect. If no such Assertion_Policy pragma exists, the policy is implementation defined.

10.a/2 **Implementation defined:** The default assertion policy.

11/2 {*AI95-00286-01*} The following language-defined library package exists:

12/5
```
{AI12-0414-1} package Ada.Assertions
    with Pure is

13/2    Assertion_Error : exception;

14/2    procedure Assert(Check : in Boolean);
        procedure Assert(Check : in Boolean; Message : in String);

15/2 end Ada.Assertions;
```

16/3 {*AI95-00286-01*} {*AI05-0290-1*} A compilation unit containing a check for an assertion (including a pragma Assert) has a semantic dependence on the Assertions library unit.

17/3 *This paragraph was deleted.*{*AI95-00286-01*} {*AI05-0290-1*}

Dynamic Semantics

18/3 {*AI95-00286-01*} {*AI05-0290-1*} If performing checks is required by the Assert assertion policy in effect at the place of a pragma Assert, the elaboration of the pragma consists of evaluating the boolean expression, and if the result is False, evaluating the Message argument, if any, and raising the exception Assertions.Assertion_Error, with a message if the Message argument is provided.

19/2 {*AI95-00286-01*} Calling the procedure Assertions.Assert without a Message parameter is equivalent to:

20/2
```
if Check = False then
   raise Ada.Assertions.Assertion_Error;
end if;
```

21/2 {*AI95-00286-01*} Calling the procedure Assertions.Assert with a Message parameter is equivalent to:

22/2
```
if Check = False then
   raise Ada.Assertions.Assertion_Error with Message;
end if;
```

23/2 {*AI95-00286-01*} The procedures Assertions.Assert have these effects independently of the assertion policy in effect.

Bounded (Run-Time) Errors

{*AI05-0274-1*} {*AI12-0439-1*} It is a bounded error to invoke a potentially blocking operation (see 9.5.1) during the evaluation of an assertion expression associated with a call on, or return from, a protected operation. If the bounded error is detected, Program_Error is raised. If not detected, execution proceeds normally, but if it is invoked within a protected action, it can result in deadlock or a (nested) protected action. 23.1/5

Implementation Requirements

{*AI12-0179-1*} {*AI12-0265-1*} Any postcondition expression, type invariant expression, or default initial condition expression occurring in the specification of a language-defined unit is enabled (see 6.1.1, 7.3.2, and 7.3.3). 23.2/5

Ramification: The Assertion_Policy does not have an effect on such postconditions, invariants, and default initial conditions. This has no execution impact since such assertions shouldn't fail anyway (see the next rule). 23.a/5

{*AI12-0179-1*} {*AI12-0265-1*} The evaluation of any such postcondition, type invariant, or default initial condition expression shall either yield True or propagate an exception from a raise_expression that appears within the assertion expression. 23.3/5

Ramification: In other words, evaluating such an assertion expression will not return a result of False, nor will it propagate an exception other than by evaluating a raise_expression which is syntactically all or part of the assertion expression. 23.b/5

To be honest: Evaluation of any expression might raise Storage_Error. 23.c/5

Reason: This allows the Reference Manual to express semantic requirements as postconditions, invariants, or default initial conditions (which are invariably clearer than English prose would be) while keeping it clear that failing the assertion check (or any other run time check) is not conforming behavior. 23.d/5

{*AI12-0112-1*} Any precondition expression occurring in the specification of a language-defined unit is enabled (see 6.1.1) unless suppressed (see 11.5). Similarly, any predicate checks for a subtype occurring in the specification of a language-defined unit are enabled (see 3.2.4) unless suppressed. 23.4/5

Reason: {*AI12-0005-1*} {*AI12-0112-1*} This allows the Reference Manual to express runtime requirements on the client of a language-defined unit as preconditions or predicates (which are clearer than English prose would be). Some such requirements can be suppressed with pragma Suppress. Ada 2012 and earlier versions did not provide a mechanism to suppress such code. 23.e/5

Implementation Permissions

{*AI95-00286-01*} Assertion_Error may be declared by renaming an implementation-defined exception from another package. 24/2

Reason: This permission is intended to allow implementations which had an implementation-defined Assert pragma to continue to use their originally defined exception. Without this permission, such an implementation would be incorrect, as Exception_Name would return the wrong name. 24.a/2

{*AI95-00286-01*} Implementations may define their own assertion policies. 25/2

{*AI05-0274-1*} {*AI12-0445-1*} If the result of a function call in an assertion is not used to determine the value of the assertion expression, an implementation is permitted to omit the function call. [This permission applies even if the function has side effects.] 26/5

{*AI05-0274-1*} {*AI12-0444-1*} An implementation may disallow the specification of an assertion expression if the evaluation of the expression has a side effect such that an immediate reevaluation of the expression can produce a different value. Similarly, an implementation may disallow the specification of an assertion expression that is checked as part of a call on or return from a callable entity *C*, if the evaluation of the expression has a side effect such that the evaluation of some other assertion expression associated with the same call of (or return from) *C* can produce a different value than in the case when the first expression had not been evaluated. 27/5

27.a/3 **Ramification:** This allows an implementation to reject such assertions. To maximize portability, assertions should not include expressions that contain these sorts of side effects.

27.b/3 **Discussion:** The intended effect of the second part of the rule (the part starting with "Similarly") is that an evaluation of the involved assertion expressions (subtype predicates, type invariants, preconditions and postconditions) in any order yields identical results.

27.c/3 The rule is intended to apply to all of the assertion expressions that are evaluated at the start of call (and similarly for the assertion expressions that are evaluated during the return from a call), but not other assertions actually given in the body, nor between the assertions checked at the start and end of the call. Specifically, a side effect that alters a variable in a function called from a precondition expression that changes the result of a postcondition expression of the same subprogram does *not* trigger these rules unless it also changes the value of a reevaluation of the precondition expression.

Language Design Principles

27.d/4 {*AI12-0005-1*} Our intent is that any assertion expression that violates this Implementation Permission is considered pathological. We definitely want compilers to be able to assume that if you evaluate an assertion expression once and it is True, you don't need to evaluate it again if all you are doing in the meantime is evaluating assertion expressions. We were unable to find wording that had this effect that didn't throw out important other cases (logging, memo functions), so we settled for a strong warning that compilers can reject such pathologies. Perhaps in a future version of Ada we'll be able to tighten this up.

28/5 NOTE {*AI95-00286-01*} {*AI12-0442-1*} {*AI12-0447-1*} Normally, the boolean expression in a pragma Assert should not call functions that have significant side effects when the result of the expression is True, so that the particular assertion policy in effect will not affect normal operation of the program.

Extensions to Ada 95

28.a/2 {*AI95-00286-01*} Pragmas Assert and Assertion_Policy, and package Assertions are new.

Incompatibilities With Ada 2005

28.b/3 {*AI05-0274-1*} There now is an Implementation Permission to reject an assertion expression that calls a function that has a side effect such that an immediate reevalution of the expression could produce a different value. This means that a pragma Assert that works in Ada 2005 might be illegal in Ada 2012 in the unlikely event that the compiler detected such an error. This should be unlikely to occur in practice and it is considered a good thing, as the original expression was tricky and probably was not portable (as order of evaluation is unspecified within an expression). Moreover, no compiler is *required* to reject such expressions, so there is no need for any compiler to change behavior.

Extensions to Ada 2005

28.c/3 {*AI05-0290-1*} Assertion_Policy pragmas are now allowed in more places and can specify behavior for individual kinds of assertions.

Wording Changes from Ada 2012

28.d/5 {*AI12-0112-1*} Added wording that preconditions and predicates given on language-defined units are always checked unless suppressed (that is, they act like language-defined checks). This is not considered an inconsistency, since there are no such preconditions or predicates in Ada 2012.

28.e/5 {*AI12-0179-1*} {*AI12-0265-1*} **Correction:** Added wording that postconditions, type invariants, and default initial conditions given on language-defined units cannot fail. This is not considered an inconsistency, since there are no such postconditions, invariants, or default initial conditions in Ada 2012.

28.f/5 {*AI12-0265-1*} Added default initial conditions to the kinds of assertions (see 7.3.3).

28.g/5 {*AI12-0396-1*} **Correction:** Added a definition of assertion aspects, used in some freezing rules (see 13.14).

11.4.3 Example of Exception Handling

Examples

1/5 {*AI12-0440-1*} Exception handling can be used to separate the detection of an error from the response to that error:

2/5
```
{AI95-00433-01} {AI12-0178-1} package File_System is
    type Data_Type is ...;
    type File_Handle is limited private;
```

```
    File_Not_Found : exception;                                                    3
    procedure Open(F : in out File_Handle; Name : String);
        -- raises File_Not_Found if named file does not exist

    End_Of_File : exception;                                                       4
    procedure Read(F : in out File_Handle; Data : out Data_Type);
        -- raises End_Of_File if the file is not open

{AI12-0178-1}     ...                                                          5/5
private
    ...
end File_System;
```

Reason: {*AI12-0178-1*} The first ... provides a place for Close to be declared, and the second ... provides a place for File_Handle to be completed. 5.a/5

```
{AI12-0178-1} package body File_System is                                      5.1/5
    ...
```

Reason: {*AI12-0178-1*} This ... provides a place for File_Exists and the body of Close to be declared. 5.b/5

```
{AI95-00433-01} {AI12-0178-1}     procedure Open(F : in out File_Handle; Name :  6/5
String) is
    begin
        if File_Exists(Name) then
            ...
        else
            raise File_Not_Found with "File not found: " & Name & ".";
        end if;
    end Open;

    procedure Read(F : in out File_Handle; Data : out Data_Type) is            7
    begin
        if F.Current_Position <= F.Last_Position then
            ...
        else
            raise End_Of_File;
        end if;
    end Read;

    ...                                                                        8
end File_System;                                                               9

{AI12-0178-1} with Ada.Text_IO;                                              10/5
with Ada.Exceptions;
with File_System; use File_System;
use Ada;
procedure Main is
    Verbosity_Desired : Boolean := ...;
begin
    ... -- call operations in File_System
exception
    when End_Of_File =>
        Close(Some_File);
    when Not_Found_Error : File_Not_Found =>
        Text_IO.Put_Line(Exceptions.Exception_Message(Not_Found_Error));
    when The_Error : others =>
        Text_IO.Put_Line("Unknown error:");
        if Verbosity_Desired then
            Text_IO.Put_Line(Exceptions.Exception_Information(The_Error));
        else
            Text_IO.Put_Line(Exceptions.Exception_Name(The_Error));
            Text_IO.Put_Line(Exceptions.Exception_Message(The_Error));
        end if;
        raise;
end Main;
```

{*AI12-0440-1*} In the above example, the File_System package contains information about detecting certain exceptional situations, but it does not specify how to handle those situations. Procedure Main specifies how to handle them; other clients of File_System can have different handlers, even though the exceptional situations arise from the same basic causes. 11/5

Wording Changes from Ada 83

11.a/3 {*AI05-0299-1*} The sections labeled “Exceptions Raised During ...” are subsumed by this subclause, and by parts of Clause 9.

11.5 Suppressing Checks

1/2 {*AI95-00224-01*} *Checking pragmas* give instructions to an implementation on handling language-defined checks. A pragma Suppress gives permission to an implementation to omit certain language-defined checks, while a pragma Unsuppress revokes the permission to omit checks.

1.a/5 **Term entry: suppress a check** — assert that the check cannot fail, and request that the compiler optimize by disabling the check
Note: The compiler is not required to honor this request. Suppressing checks that can fail can cause a program to behave in arbitrary ways.

2/3 {*AI05-0264-1*} A *language-defined check* (or simply, a “check”) is one of the situations defined by this document that requires a check to be made at run time to determine whether some condition is true. A check *fails* when the condition being checked is False, causing an exception to be raised.

2.a **Discussion:** All such checks are defined under “Dynamic Semantics” in clauses and subclauses throughout the standard.

2.b/5 **Term entry: check** — test made during execution to determine whether a language rule has been violated

Syntax

3/2 {*AI95-00224-01*} The forms of checking pragmas are as follows:

4/2 {*AI95-00224-01*} **pragma** Suppress(identifier);

4.1/2 {*AI95-00224-01*} **pragma** Unsuppress(identifier);

5/2 {*AI95-00224-01*} A checking pragma is allowed only immediately within a declarative_part, immediately within a package_specification, or as a configuration pragma.

Legality Rules

6/2 {*AI95-00224-01*} The identifier shall be the name of a check.

7/2 *This paragraph was deleted.*{*AI95-00224-01*}

Static Semantics

7.1/2 {*AI95-00224-01*} A checking pragma applies to the named check in a specific region, and applies to all entities in that region. A checking pragma given in a declarative_part or immediately within a package_specification applies from the place of the pragma to the end of the innermost enclosing declarative region. The region for a checking pragma given as a configuration pragma is the declarative region for the entire compilation unit (or units) to which it applies.

7.2/3 {*AI95-00224-01*} {*AI05-0229-1*} {*AI05-0290-1*} If a checking pragma applies to a generic_instantiation, then the checking pragma also applies to the entire instance.

7.a/3 **Ramification:** {*AI05-0290-1*} This means that a Suppress pragma that occurs in a scope enclosing the declaration of a generic unit but not also enclosing the declaration of a given instance of that generic unit will not apply to constructs within the given instance.

8/2 {*AI95-00224-01*} A pragma Suppress gives permission to an implementation to omit the named check (or every check in the case of All_Checks) for any entities to which it applies. If permission has been given to suppress a given check, the check is said to be *suppressed*.

8.a **Ramification:** A check is suppressed even if the implementation chooses not to actually generate better code. This allows the implementation to raise Program_Error, for example, if the erroneousness is detected.

{*AI95-00224-01*} A pragma Unsuppress revokes the permission to omit the named check (or every check in the case of All_Checks) given by any pragma Suppress that applies at the point of the pragma Unsuppress. The permission is revoked for the region to which the pragma Unsuppress applies. If there is no such permission at the point of a pragma Unsuppress, then the pragma has no effect. A later pragma Suppress can renew the permission. 8.1/2

The following are the language-defined checks: 9

- {*AI12-0309-1*} [The following checks correspond to situations in which the exception Constraint_Error is raised upon failure of a language-defined check.] 10/5

  {*8652/0036*} {*AI95-00176-01*} {*AI95-00231-01*} Access_Check 11/2
  : When evaluating a dereference (explicit or implicit), check that the value of the name is not **null**. When converting to a subtype that excludes null, check that the converted value is not **null**.

  Discriminant_Check 12
  : Check that the discriminants of a composite value have the values imposed by a discriminant constraint. Also, when accessing a record component, check that it exists for the current discriminant values.

  {*AI95-00434-01*} Division_Check 13/2
  : Check that the second operand is not zero for the operations /, **rem** and **mod**.

  Index_Check 14
  : Check that the bounds of an array value are equal to the corresponding bounds of an index constraint. Also, when accessing a component of an array object, check for each dimension that the given index value belongs to the range defined by the bounds of the array object. Also, when accessing a slice of an array object, check that the given discrete range is compatible with the range defined by the bounds of the array object.

  Length_Check 15
  : Check that two arrays have matching components, in the case of array subtype conversions, and logical operators for arrays of boolean components.

  Overflow_Check 16
  : Check that a scalar value is within the base range of its type, in cases where the implementation chooses to raise an exception instead of returning the correct mathematical result.

  {*AI12-0244-1*} Range_Check 17/5
  : Check that a scalar value satisfies a range constraint. Also, for the elaboration of a subtype_indication, check that the constraint (if present) is compatible with the subtype denoted by the subtype_mark. Also, for an aggregate, check that an index or discriminant value belongs to the corresponding subtype. Also, check that when the result of an operation yields an array, the value of each component belongs to the component subtype. Also, for the attributes Value, Wide_Value, and Wide_Wide_Value, check that the given string has the appropriate syntax and value for the base subtype of the prefix of the attribute_reference.

  Tag_Check 18
  : Check that operand tags in a dispatching call are all equal. Check for the correct tag on tagged type conversions, for an assignment_statement, and when returning a tagged limited object from a function.

- {*AI12-0309-1*} [The following checks correspond to situations in which the exception Program_Error is raised upon failure of a language-defined check.] 19/5

  {*AI95-00280*} Accessibility_Check 19.1/2
  : Check the accessibility level of an entity or view.

19.2/2 {*AI95-00280*} Allocation_Check
For an allocator, check that the master of any tasks to be created by the allocator is not yet completed or some dependents have not yet terminated, and that the finalization of the collection has not started.

20 Elaboration_Check
When a subprogram or protected entry is called, a task activation is accomplished, or a generic instantiation is elaborated, check that the body of the corresponding unit has already been elaborated.

21/5 {*AI95-00280*} {*AI12-0309-1*} Program_Error_Check
Other language-defined checks that raise Program_Error: that subtypes with predicates are not used to index an array in a generic unit; that the maximum number of chunks is greater than zero; that the default value of an out parameter is convertible; that there is no misuse of functions in a generic with a class-wide actual type; that there are not colliding External_Tag values; that there is no misuse of operations of unchecked union types.

22/5 • {*AI12-0309-1*} [The following check corresponds to situations in which the exception Storage_Error is raised upon failure of a language-defined check.]

23 Storage_Check
Check that evaluation of an allocator does not require more space than is available for a storage pool. Check that the space available for a task or subprogram has not been exceeded.

23.a **Reason:** We considered splitting this out into three categories: Pool_Check (for allocators), Stack_Check (for stack usage), and Heap_Check (for implicit use of the heap — use of the heap other than through an allocator). Storage_Check would then represent the union of these three. However, there seems to be no compelling reason to do this, given that it is not feasible to split Storage_Error.

23.1/5 • {*AI12-0309-1*} [The following check corresponds to situations in which the exception Tasking_Error is raised upon failure of a language-defined check.]

23.2/5 {*AI12-0309-1*} Tasking_Check
Check that all tasks activated successfully. Check that a called task has not yet terminated.

23.3/5 • {*AI12-0112-1*} {*AI12-0311-1*} [The following checks correspond to situations in which the exception Assertion_Error is raised upon failure of a language-defined check.] For a language-defined unit *U* associated with one of these checks in the list below, the check refers to performance of checks associated with the Pre, Static_Predicate, and Dynamic_Predicate aspects associated with any entity declared in a descendant of *U*, or in an instance of a generic unit which is, or is declared in, a descendant of *U*. Each check is associated with one or more units:

23.4/5 {*AI12-0311-1*} Calendar_Assertion_Check
Calendar.

23.5/5 {*AI12-0311-1*} Characters_Assertion_Check
Characters, Wide_Characters, and Wide_Wide_Characters.

23.6/5 {*AI12-0112-1*} {*AI12-0311-1*} Containers_Assertion_Check
Containers.

23.7/5 {*AI12-0311-1*} Interfaces_Assertion_Check
Interfaces.

23.8/5 {*AI12-0311-1*} IO_Assertion_Check
Sequential_IO, Direct_IO, Text_IO, Wide_Text_IO, Wide_Wide_Text_IO, Storage_IO, Streams.Stream_IO, and Directories.

23.9/5 {*AI12-0311-1*} Numerics_Assertion_Check
Numerics.

{*AI12-0311-1*} Strings_Assertion_Check 23.10/5
Strings.

{*AI12-0311-1*} System_Assertion_Check 23.11/5
System.

Ramification: {*AI12-0311-1*} Any unit *U* is a descendant of itself. 23.b/5

Reason: {*AI12-0112-1*} One could use **pragma** Assertion_Policy to eliminate such checks, but that would require recompiling the language defined packages (the assertion policy that determines whether the assertion checks are made is that used to compile the unit). In addition, we do not want to specify the behavior of language-defined operations if the precondition fails; that is different than the usual behavior of Assertion_Policy. By using Suppress for this purpose, we make it clear that a failed check that is suppressed means erroneous execution. 23.c/5

To be honest: {*AI12-0311-1*} The preceding rule about an instance of a generic where the generic is declared in *U* really extends recursively to handle the case of a generic package `G1` which declares another generic package `G1.G2`, which declares another generic package `G1.G2.G3`, and so on. So if `G1` is declared in some predefined unit *U* then, for purposes of defining these checks, `Some_Instance_Of_G1.G2` is also considered to be declared in *U*. 23.d/5

- {*AI12-0309-1*} [The following check corresponds to all situations in which any predefined exception is raised upon failure of a check.] 24/5

  {*AI05-0290-1*} All_Checks 25/3
  Represents the union of all checks; suppressing All_Checks suppresses all checks other than those associated with assertions. In addition, an implementation is allowed (but not required) to behave as if a pragma Assertion_Policy(Ignore) applies to any region to which pragma Suppress(All_Checks) applies.

  Ramification: {*AI12-0309-1*} All_Checks may include implementation-defined checks. It does not include, however, explicit raises of predefined exceptions (including those mandated for language constructs), nor those propagated from language-defined subprograms. 25.a/5

  To be honest: {*AI05-0005-1*} There are additional checks defined in various Specialized Needs Annexes that are not listed here. Nevertheless, they are included in All_Checks and named in a Suppress pragma on implementations that support the relevant annex. Look up “check, language-defined” in the index to find the complete list. 25.b/3

  Discussion: {*AI05-0290-1*} We don't want to say that assertions are suppressed, because we don't want the potential failure of an assertion to cause erroneous execution (see below). Thus they are excluded from the suppression part of the above rule and then handled with an implicit Ignore policy. 25.c/3

Erroneous Execution

{*AI12-0112-1*} {*AI12-0311-1*} If a given check has been suppressed, and the corresponding error situation occurs, the execution of the program is erroneous. Similarly, if a precondition check has been suppressed and the evaluation of the precondition would have raised an exception, execution is erroneous. 26/5

Reason: {*AI12-0112-1*} It's unclear that a precondition expression that executes **`raise`** *`some_exception`* is an "error situation"; the precondition never actually evaluates to False in that case. Thus, we spell out that case. We only allow suppressing preconditions associated with language-defined units; other preconditions follow the rules of the appropriate Assertion_Policy. 26.a/5

Implementation Permissions

{*AI95-00224-01*} An implementation is allowed to place restrictions on checking pragmas, subject only to the requirement that pragma Unsuppress shall allow any check names supported by pragma Suppress. An implementation is allowed to add additional check names, with implementation-defined semantics. When Overflow_Check has been suppressed, an implementation may also suppress an unspecified subset of the Range_Checks. 27/2

This paragraph was deleted.{*AI95-00224-01*} 27.a/2

Implementation defined: Implementation-defined check names. 27.b

Discussion: For Overflow_Check, the intention is that the implementation will suppress any Range_Checks that are implemented in the same manner as Overflow_Checks (unless they are free). 27.c

27.1/2 {*AI95-00224-01*} An implementation may support an additional parameter on pragma Unsuppress similar to the one allowed for pragma Suppress (see J.10). The meaning of such a parameter is implementation-defined.

27.c.1/2 **Implementation defined:** Existence and meaning of second parameter of pragma Unsuppress.

Implementation Advice

28 The implementation should minimize the code executed for checks that have been suppressed.

28.a.1/2 **Implementation Advice:** Code executed for checks that have been suppressed should be minimized.

28.a **Implementation Note:** However, if a given check comes for free (for example, the hardware automatically performs the check in parallel with doing useful work) or nearly free (for example, the check is a tiny portion of an expensive run-time system call), the implementation should not bother to suppress the check. Similarly, if the implementation detects the failure at compile time and provides a warning message, there is no need to actually suppress the check.

29/5 NOTE 1 {*AI12-0442-1*} There is no guarantee that a suppressed check is actually removed; hence a pragma Suppress is useful only to improve efficiency.

29.1/2 NOTE 2 {*AI95-00224-01*} It is possible to give both a pragma Suppress and Unsuppress for the same check immediately within the same declarative_part. In that case, the last pragma given determines whether or not the check is suppressed. Similarly, it is possible to resuppress a check which has been unsuppressed by giving a pragma Suppress in an inner declarative region.

Examples

30/2 {*AI95-00224-01*} *Examples of suppressing and unsuppressing checks:*

31/2
```
{AI95-00224-01} pragma Suppress(Index_Check);
pragma Unsuppress(Overflow_Check);
```

Extensions to Ada 83

31.a A pragma Suppress is allowed as a configuration pragma. A pragma Suppress without a name is allowed in a package_specification.

31.b Additional check names are added. We allow implementations to define their own checks.

Wording Changes from Ada 83

31.c We define the checks in a distributed manner. Therefore, the long list of what checks apply to what is merely a NOTE.

31.d We have removed the detailed rules about what is allowed in a pragma Suppress, and allow implementations to invent their own. The RM83 rules weren't quite right, and such a change is necessary anyway in the presence of implementation-defined checks.

31.e We make it clear that the difference between a Range_Check and an Overflow_Check is fuzzy. This was true in Ada 83, given RM83-11.6, but it was not clear. We considered removing Overflow_Check from the language or making it obsolescent, just as we did for Numeric_Error. However, we kept it for upward compatibility, and because it may be useful on machines where range checking costs more than overflow checking, but overflow checking still costs something. Different compilers will suppress different checks when asked to suppress Overflow_Check — the nonuniformity in this case is not harmful, and removing it would have a serious impact on optimizers.

31.f Under Access_Check, dereferences cover the cases of selected_component, indexed_component, slice, and attribute that are listed in RM83, as well as the new explicit_dereference, which was included in selected_component in RM83.

Extensions to Ada 95

31.g/2 {*AI95-00224-01*} Pragma Unsuppress is new.

31.h/2 {*AI95-00280-01*} Allocation_Check was added to support suppressing the new check on allocators (see 4.8).

Wording Changes from Ada 95

31.i/2 {*8652/0036*} {*AI95-00176-01*} {*AI95-00224-01*} The description of Access_Check was corrected by the Corrigendum to include the discriminant case. This change was then replaced by the more general notion of checking conversions to subtypes that exclude null in Ada 2005.

31.j/2 {*AI95-00224-01*} The On parameter of pragma Suppress was moved to Annex J (see J.10). This feature's effect is inherently nonportable, depending on the implementation's model of computation. Compiler surveys demonstrated this,

showing that implementations vary widely in the interpretation of these parameters, even on the same target. While this is relatively harmless for Suppress (which is never required to do anything), it would be a significant problem for Unsuppress (we want the checks to be made for all implementations). By moving it, we avoid needing to define the meaning of Unsuppress with an On parameter.

{*AI95-00280-01*} The order of the Program_Error checks was corrected to be alphabetical. 31.k/2

Wording Changes from Ada 2005

{*AI05-0290-1*} {*AI12-0005-1*} The effect of a checking pragma no longer applies inside an inlined subprogram body. While this could change the behavior of a program that depends on a check being suppressed in an inlined body, such a program is erroneous and thus no behavior can be depended upon anyway. It's also likely to be very rare. We make this change so that inlining has no effect on the meaning of the subprogram body (since inlining is never required, this is necessary in order to be able to reason about the body), and so that assertion policies and suppress work the same way for inlining. 31.l/4

Extensions to Ada 2012

{*AI12-0005-1*} {*AI12-0112-1*} {*AI12-0311-1*} The various assertion checks for language-defined packages, such as Calendar_Check and Container_Check, are new. 31.m/5

{*AI12-0309-1*} Program_Error_Check and Tasking_Check are new; all core language-defined checks are now covered with check names. 31.n/5

Wording Changes from Ada 2012

{*AI12-0244-1*} **Correction**: Range_Check is defined to include checks associated with the Value and related attributes. 31.o/5

11.6 Exceptions and Optimization

{*AI05-0299-1*} [This subclause gives permission to the implementation to perform certain "optimizations" that do not necessarily preserve the canonical semantics.] 1/3

Dynamic Semantics

{*AI05-0299-1*} The rest of this Reference Manual (outside this subclause) defines the *canonical semantics* of the language. [The canonical semantics of a given (legal) program determines a set of possible external effects that can result from the execution of the program with given inputs.] 2/3

Ramification: Note that the canonical semantics is a set of possible behaviors, since some reordering, parallelism, and nondeterminism is allowed by the canonical semantics. 2.a

Discussion: {*AI05-0299-1*} The following parts of the canonical semantics are of particular interest to the reader of this subclause: 2.b/3

- Behavior in the presence of abnormal objects and objects with invalid representations (see 13.9.1). 2.c
- Various actions that are defined to occur in an arbitrary order. 2.d
- Behavior in the presence of a misuse of Unchecked_Deallocation, Unchecked_Access, or imported or exported entity (see Clause 13). 2.e

{*AI05-0299-1*} [As explained in 1.1.3, "Conformity of an Implementation", the external effect of a program is defined in terms of its interactions with its external environment. Hence, the implementation can perform any internal actions whatsoever, in any order or in parallel, so long as the external effect of the execution of the program is one that is allowed by the canonical semantics, or by the rules of this subclause.] 3/3

Ramification: Note that an optimization can change the external effect of the program, so long as the changed external effect is an external effect that is allowed by the semantics. Note that the canonical semantics of an erroneous execution allows any external effect whatsoever. Hence, if the implementation can prove that program execution will be erroneous in certain circumstances, there need not be any constraints on the machine code executed in those circumstances. 3.a

Implementation Permissions

4 The following additional permissions are granted to the implementation:

5/5 - {*AI12-0444-1*} An implementation can omit raising an exception when a language-defined check fails. Instead, the operation that failed the check can simply yield an *undefined result*. The exception is required to be raised by the implementation only if, in the absence of raising it, the value of this undefined result would have some effect on the external interactions of the program. In determining this, the implementation shall not presume that an undefined result has a value that belongs to its subtype, nor even to the base range of its type, if scalar. [Having removed the raise of the exception, the canonical semantics will in general allow the implementation to omit the code for the check, and some or all of the operation itself.]

5.a **Ramification:** Even without this permission, an implementation can always remove a check if it cannot possibly fail.

5.b **Reason:** We express the permission in terms of removing the raise, rather than the operation or the check, as it minimizes the disturbance to the canonical semantics (thereby simplifying reasoning). By allowing the implementation to omit the raise, it thereby does not need to "look" at what happens in the exception handler to decide whether the optimization is allowed.

5.c **Discussion:** The implementation can also omit checks if they cannot possibly fail, or if they could only fail in erroneous executions. This follows from the canonical semantics.

5.d **Implementation Note:** This permission is intended to allow normal "dead code removal" optimizations, even if some of the removed code might have failed some language-defined check. However, one may not eliminate the raise of an exception if subsequent code presumes in some way that the check succeeded. For example:

5.e
```
      if X * Y > Integer'Last then
          Put_Line("X * Y overflowed");
      end if;
    exception
      when others =>
          Put_Line("X * Y overflowed");
```

5.e.1 If X*Y does overflow, you may not remove the raise of the exception if the code that does the comparison against Integer'Last presumes that it is comparing it with an in-range Integer value, and hence always yields False.

5.f As another example where a raise may not be eliminated:

5.g
```
      subtype Str10 is String(1..10);
      type P10 is access Str10;
      X : P10 := null;
    begin
      if X.all'Last = 10 then
          Put_Line("Oops");
      end if;
```

5.g.1 In the above code, it would be wrong to eliminate the raise of Constraint_Error on the "X.all" (since X is null), if the code to evaluate 'Last always yields 10 by presuming that X.all belongs to the subtype Str10, without even "looking".

6/5 - {*AI05-0229-1*} {*AI12-0445-1*} If an exception is raised due to the failure of a language-defined check, then upon reaching the corresponding exception_handler (or the termination of the task, if none), the external interactions that have occurred have to reflect only that the exception was raised somewhere within the execution of the sequence_of_statements with the handler (or the task_body), possibly earlier (or later if the interactions are independent of the result of the checked operation) than that defined by the canonical semantics, but not within the execution of some abort-deferred operation or *independent* subprogram that does not dynamically enclose the execution of the construct whose check failed. An independent subprogram is one that is defined outside the library unit containing the construct whose check failed, and for which the Inline aspect is False. Any assignment that occurred outside of such abort-deferred operations or independent subprograms can be disrupted by the raising of the exception, causing the object or its parts to become abnormal, and certain subsequent uses of the object to be erroneous, as explained in 13.9.1.

6.a **Reason:** We allow such variables to become abnormal so that assignments (other than to atomic variables) can be disrupted due to “imprecise” exceptions or instruction scheduling, and so that assignments can be reordered so long as the correct results are produced in the end if no language-defined checks fail.

Ramification: If a check fails, no result dependent on the check may be incorporated in an external interaction. In other words, there is no permission to output meaningless results due to postponing a check. 6.b

Discussion: We believe it is important to state the extra permission to reorder actions in terms of what the programmer can expect at run time, rather than in terms of what the implementation can assume, or what transformations the implementation can perform. Otherwise, how can the programmer write reliable programs? 6.c

{*AI05-0299-1*} This subclause has two conflicting goals: to allow as much optimization as possible, and to make program execution as predictable as possible (to ease the writing of reliable programs). The rules given above represent a compromise. 6.d/3

Consider the two extremes: 6.e

{*AI05-0299-1*} The extreme conservative rule would be to delete this subclause entirely. The semantics of Ada would be the canonical semantics. This achieves the best predictability. It sounds like a disaster from the efficiency point of view, but in practice, implementations would provide modes in which less predictability but more efficiency would be achieved. Such a mode could even be the out-of-the-box mode. In practice, implementers would provide a compromise based on their customer's needs. Therefore, we view this as one viable alternative. 6.f/3

The extreme liberal rule would be “the language does not specify the execution of a program once a language-defined check has failed; such execution can be unpredictable”. This achieves the best efficiency. It sounds like a disaster from the predictability point of view, but in practice it might not be so bad. A user would have to assume that exception handlers for exceptions raised by language-defined checks are not portable. They would have to isolate such code (like all nonportable code), and would have to find out, for each implementation of interest, what behaviors can be expected. In practice, implementations would tend to avoid going so far as to punish their customers too much in terms of predictability. 6.g

{*AI05-0299-1*} The most important thing about this subclause is that users understand what they can expect at run time, and implementers understand what optimizations are allowed. Any solution that makes this subclause contain rules that can interpreted in more than one way is unacceptable. 6.h/3

We have chosen a compromise between the extreme conservative and extreme liberal rules. The current rule essentially allows arbitrary optimizations within a library unit and inlined subprograms reachable from it, but disallow semantics-disrupting optimizations across library units in the absence of inlined subprograms. This allows a library unit to be debugged, and then reused with some confidence that the abstraction it manages cannot be broken by bugs outside the library unit. 6.i

NOTE {*AI05-0299-1*} The permissions granted by this subclause can have an effect on the semantics of a program only if the program fails a language-defined check. 7/3

Wording Changes from Ada 83

RM83-11.6 was unclear. It has been completely rewritten here; we hope this version is clearer. Here's what happened to each paragraph of RM83-11.6: 7.a

- Paragraphs 1 and 2 contain no semantics; they are merely pointing out that anything goes if the canonical semantics is preserved. We have similar introductory paragraphs, but we have tried to clarify that these are not granting any “extra” permission beyond what the rest of the document allows. 7.b
- Paragraphs 3 and 4 are reflected in the “extra permission to reorder actions”. Note that this permission now allows the reordering of assignments in many cases. 7.c
- Paragraph 5 is moved to 4.5, “Operators and Expression Evaluation”, where operator association is discussed. Hence, this is no longer an “extra permission” but is part of the canonical semantics. 7.d
- Paragraph 6 now follows from the general permission to store out-of-range values for unconstrained subtypes. Note that the parameters and results of all the predefined operators of a type are of the unconstrained subtype of the type. 7.e
- Paragraph 7 is reflected in the “extra permission to avoid raising exceptions”. 7.f

{*AI05-0299-1*} {*AI12-0005-1*} We moved subclause 11.5, “Suppressing Checks” (which was numbered 11.7) from after 11.6 to before 11.6, in order to preserve the famous number “11.6” (given the changes to earlier subclauses in Clause 11). 7.g/5

12 Generic Units

A *generic unit* is a program unit that is either a generic subprogram or a generic package. A generic unit is a *template*[, which can be parameterized, and from which corresponding (nongeneric) subprograms or packages can be obtained]. The resulting program units are said to be *instances* of the original generic unit. 1

Term entry: generic unit — template for a (nongeneric) program unit 1.a/5
Note 1: The template can be parameterized by objects, types, subprograms, and packages.
Note 2: Generic units can be used to perform the role that macros sometimes play in other languages.

Term entry: generic instance — nongeneric unit created by the instantiation of a generic unit 1.b/5

[A generic unit is declared by a generic_declaration. This form of declaration has a generic_formal_part declaring any generic formal parameters. An instance of a generic unit is obtained as the result of a generic_instantiation with appropriate generic actual parameters for the generic formal parameters. An instance of a generic subprogram is a subprogram. An instance of a generic package is a package. 2

Generic units are templates. As templates they do not have the properties that are specific to their nongeneric counterparts. For example, a generic subprogram can be instantiated but it cannot be called. In contrast, an instance of a generic subprogram is a (nongeneric) subprogram; hence, this instance can be called but it cannot be used to produce further instances.] 3

12.1 Generic Declarations

[A generic_declaration declares a generic unit, which is either a generic subprogram or a generic package. A generic_declaration includes a generic_formal_part declaring any generic formal parameters. A generic formal parameter can be an object; alternatively (unlike a parameter of a subprogram), it can be a type, a subprogram, or a package.] 1

Syntax

generic_declaration ::= 2
  generic_subprogram_declaration | generic_package_declaration

{*AI05-0183-1*} generic_subprogram_declaration ::= 3/3
  generic_formal_part subprogram_specification
    [aspect_specification];

generic_package_declaration ::= 4
  generic_formal_part package_specification;

Ramification: {*AI05-0183-1*} No syntax change is needed here to allow an aspect_specification; a generic package can have an aspect_specification because a package_specification allows an aspect_specification. 4.a/3

generic_formal_part ::= 5
  generic {generic_formal_parameter_declaration | use_clause}

generic_formal_parameter_declaration ::= 6
  formal_object_declaration
 | formal_type_declaration
 | formal_subprogram_declaration
 | formal_package_declaration

The only form of subtype_indication allowed within a generic_formal_part is a subtype_mark [(that is, the subtype_indication shall not include an explicit constraint)]. The defining name of a generic subprogram shall be an identifier [(not an operator_symbol)]. 7

7.a **Reason:** The reason for forbidding constraints in subtype_indications is that it simplifies the elaboration of generic_declarations (since there is nothing to evaluate), and that it simplifies the matching rules, and makes them more checkable at compile time.

Static Semantics

8/2 {*AI95-00434-01*} A generic_declaration declares a generic unit — a generic package, generic procedure, or generic function, as appropriate.

9 An entity is a *generic formal* entity if it is declared by a generic_formal_parameter_declaration. "Generic formal", or simply "formal", is used as a prefix in referring to objects, subtypes (and types), functions, procedures and packages, that are generic formal entities, as well as to their respective declarations. [Examples: "generic formal procedure" or a "formal integer type declaration".]

9.1/5 {*AI12-0371-1*} The list of generic_formal_parameter_declarations of a generic_formal_part form a *declaration list* of the generic unit.

9.a/5 **Ramification:** Aspect specifications (see 13.1.1) given in a generic formal part can only use declarations given in the formal part, and not those in the visible part of the generic unit.

Dynamic Semantics

10 The elaboration of a generic_declaration has no effect.

11 NOTE 1 Outside a generic unit a name that denotes the generic_declaration denotes the generic unit. In contrast, within the declarative region of the generic unit, a name that denotes the generic_declaration denotes the current instance.

11.a **Proof:** This is stated officially as part of the "current instance" rule in 8.6, "The Context of Overload Resolution". See also 12.3, "Generic Instantiation".

12 NOTE 2 Within a generic subprogram_body, the name of this program unit acts as the name of a subprogram. Hence this name can be overloaded, and it can appear in a recursive call of the current instance. For the same reason, this name cannot appear after the reserved word **new** in a (recursive) generic_instantiation.

13/5 NOTE 3 {*AI12-0447-1*} A default_expression or default_name appearing in a generic_formal_part is not evaluated during elaboration of the generic_formal_part; instead, it is evaluated when used. However, the usual visibility rules apply to any name used in a default, with name resolution performed based on the location of the name within the generic_formal_part.

Examples

14 *Examples of generic formal parts:*

15
```
generic      -- parameterless
```

16
```
generic
   Size : Natural;  -- formal object
```

17
```
generic
   Length : Integer := 200;          -- formal object with a default expression
```

18
```
   Area   : Integer := Length*Length; -- formal object with a default expression
```

19
```
generic
   type Item  is private;                          -- formal type
   type Index is (<>);                             -- formal type
   type Row   is array(Index range <>) of Item;    -- formal type
   with function "<"(X, Y : Item) return Boolean;     -- formal subprogram
```

20 *Examples of generic declarations declaring generic subprograms Exchange and Squaring:*

21
```
generic
   type Elem is private;
procedure Exchange(U, V : in out Elem);
```

22/5
```
{AI12-0178-1} generic
   type Item (<>) is private;
   with function "*"(U, V : Item) return Item is <>;
function Squaring(X : Item) return Item;
```

Example of a generic declaration declaring a generic package: 23

```
generic                                                                    24
   type Item   is private;
   type Vector is array (Positive range <>) of Item;
   with function Sum(X, Y : Item) return Item;
package On_Vectors is
   function Sum  (A, B : Vector) return Vector;
   function Sigma(A    : Vector) return Item;
   Length_Error : exception;
end On_Vectors;
```

Extensions to Ada 83

The syntax rule for generic_formal_parameter_declaration is modified to allow the reserved words **tagged** and **abstract**, to allow formal derived types, and to allow formal packages. 24.a

Use_clauses are allowed in generic_formal_parts. This is necessary in order to allow a use_clause within a formal part to provide direct visibility of declarations within a generic formal package. 24.b

Wording Changes from Ada 83

{*AI05-0299-1*} The syntax for generic_formal_parameter_declaration and formal_type_definition is split up into more named categories. The rules for these categories are moved to the appropriate subclauses. The names of the categories are changed to be more intuitive and uniform. For example, we changed generic_parameter_declaration to generic_formal_parameter_declaration, because the thing it declares is a generic formal, not a generic. In the others, we abbreviate "generic_formal" to just "formal". We can't do that for generic_formal_parameter_declaration, because of confusion with normal formal parameters of subprograms. 24.c/3

Extensions to Ada 2005

{*AI05-0183-1*} An optional aspect_specification can be used in a generic_subprogram_declaration (as well as a generic_package_declaration). This is described in 13.1.1. 24.d/3

Wording Changes from Ada 2012

{*AI12-0371-1*} Defined a formal part as a declaration list, so that the visibility of entities in aspect specifications is properly defined. 24.e/5

12.2 Generic Bodies

The body of a generic unit (a *generic body*) [is a template for the instance bodies. The syntax of a generic body is identical to that of a nongeneric body]. 1

Ramification: We also use terms like "generic function body" and "nongeneric package body". 1.a

Dynamic Semantics

The elaboration of a generic body has no other effect than to establish that the generic unit can from then on be instantiated without failing the Elaboration_Check. If the generic body is a child of a generic package, then its elaboration establishes that each corresponding declaration nested in an instance of the parent (see 10.1.1) can from then on be instantiated without failing the Elaboration_Check. 2

NOTE The syntax of generic subprograms implies that a generic subprogram body is always the completion of a declaration. 3

Examples

Example of a generic procedure body: 4

```
procedure Exchange(U, V : in out Elem) is  -- see 12.1                     5
   T : Elem;  -- the generic formal type
begin
   T := U;
   U := V;
   V := T;
end Exchange;
```

6 *Example of a generic function body:*

7
```
function Squaring(X : Item) return Item is  -- see 12.1
begin
   return X*X;  -- the formal operator "*"
end Squaring;
```

8 *Example of a generic package body:*

9
```
package body On_Vectors is  -- see 12.1
```

10
```
   function Sum(A, B : Vector) return Vector is
      Result : Vector(A'Range); -- the formal type Vector
      Bias   : constant Integer := B'First - A'First;
   begin
      if A'Length /= B'Length then
         raise Length_Error;
      end if;
```

11
```
      for N in A'Range loop
         Result(N) := Sum(A(N), B(N + Bias)); -- the formal function Sum
      end loop;
      return Result;
   end Sum;
```

12
```
   function Sigma(A : Vector) return Item is
      Total : Item := A(A'First); -- the formal type Item
   begin
      for N in A'First + 1 .. A'Last loop
         Total := Sum(Total, A(N)); -- the formal function Sum
      end loop;
      return Total;
   end Sigma;
end On_Vectors;
```

12.3 Generic Instantiation

1 [An instance of a generic unit is declared by a generic_instantiation.]

Language Design Principles

1.a/5 {*AI12-0300-1*} The legality of an instance should be determinable without looking at the generic body. Likewise, the legality of a generic body should be determinable without looking at any instances. Thus, the generic_declaration forms a contract between the body and the instances; if each obeys the rules with respect to the generic_declaration, then no legality problems will arise. This is really a special case of the "legality determinable via semantic dependences" Language Design Principle (see Clause 10), given that a generic_instantiation does not depend semantically upon the generic body, nor vice versa.

1.b Run-time issues are another story. For example, whether parameter passing is by copy or by reference is determined in part by the properties of the generic actuals, and thus cannot be determined at compile time of the generic body. Similarly, the contract model does not apply to Post-Compilation Rules.

Syntax

2/3 {*AI95-00218-03*} {*AI05-0183-1*} generic_instantiation ::=
package defining_program_unit_name **is**
new *generic_package*_name [generic_actual_part]
[aspect_specification];
| [overriding_indicator]
procedure defining_program_unit_name **is**
new *generic_procedure*_name [generic_actual_part]
[aspect_specification];
| [overriding_indicator]
function defining_designator **is**
new *generic_function*_name [generic_actual_part]

[aspect_specification];

generic_actual_part ::= 3
(generic_association {, generic_association})

generic_association ::= 4
[*generic_formal_parameter_*selector_name =>] explicit_generic_actual_parameter

explicit_generic_actual_parameter ::= expression | *variable_*name 5
| *subprogram_*name | *entry_*name | subtype_mark
| *package_instance_*name

A generic_association is *named* or *positional* according to whether or not the *generic_formal_-parameter_*selector_name is specified. Any positional associations shall precede any named associations. 6

{*AI05-0004-1*} {*AI12-0205-1*} The *generic actual parameter* is either the explicit_generic_actual_parameter given in a generic_association for each formal, or the corresponding default_expression, *default_*subtype_mark, or default_name if no generic_association is given for the formal. When the meaning is clear from context, the term "generic actual", or simply "actual", is used as a synonym for "generic actual parameter" and also for the view denoted by one, or the value of one. 7/5

Ramification: {*AI12-0205-1*} Any matching or other Legality Rules that apply to a generic actual are applied to any default_expression, *default_*subtype_mark, or default_name that are used as an actual. 7.a/5

Legality Rules

In a generic_instantiation for a particular kind of program unit [(package, procedure, or function)], the name shall denote a generic unit of the corresponding kind [(generic package, generic procedure, or generic function, respectively)]. 8

{*AI05-0118-1*} The *generic_formal_parameter_*selector_name of a named generic_association shall denote a generic_formal_parameter_declaration of the generic unit being instantiated. If two or more formal subprograms have the same defining name, then named associations are not allowed for the corresponding actuals. 9/3

{*AI05-0118-1*} The generic_formal_parameter_declaration for a positional generic_association is the parameter with the corresponding position in the generic_formal_part of the generic unit being instantiated. 9.1/3

{*AI12-0205-1*} A generic_instantiation shall contain at most one generic_association for each formal. Each formal without an association shall have a default_expression, *default_*subtype_mark, or subprogram_default. 10/5

{*AI12-0418-1*} In a generic unit, Legality Rules are enforced at compile time of the generic_declaration and generic body, given the properties of the formals. In the visible part and formal part of an instance, Legality Rules are enforced at compile time of the generic_instantiation, given the properties of the actuals. In other parts of an instance, Legality Rules are not enforced; this rule does not apply when a given rule explicitly specifies otherwise. 11/5

Reason: {*AI95-00114-01*} Since rules are checked using the properties of the formals, and since these properties do not always carry over to the actuals, we need to check the rules again in the visible part of the instance. For example, only if a tagged type is limited may an extension of it have limited components in the record_extension_part. A formal tagged limited type is limited, but the actual might be nonlimited. Hence any rule that requires a tagged type to be limited runs into this problem. Such rules are rare; in most cases, the rules for matching of formals and actuals guarantee that if the rule is obeyed in the generic unit, then it has to be obeyed in the instance. 11.a/2

11.a.1/3 {*AI05-0005-1*} Ada 2012 addendum: Such Legality Rules are not as rare as the authors of Ada 95 hoped; there are more than 30 of them known at this point. They are indexed under "generic contract issue" and are associated with the boilerplate "In addition to the places where Legality Rules normally apply...". Indeed, there is only one known rule where rechecking in the specification is needed and where rechecking in the private part is *not* wanted (it is in 3.4, but even it needs rechecking when tagged types are involved).

11.b **Ramification:** The "properties" of the formals are determined without knowing anything about the actuals:

11.c/1 - {*8652/0095*} {*AI95-00034-01*} A formal derived subtype is constrained if and only if the ancestor subtype is constrained. A formal array type is constrained if and only if the declarations say so. A formal private type is constrained if it does not have a discriminant part. Other formal subtypes are unconstrained, even though they might be constrained in an instance.

11.d - A formal subtype can be indefinite, even though the copy might be definite in an instance.

11.e - A formal object of mode **in** is not a static constant; in an instance, the copy is static if the actual is.

11.f - A formal subtype is not static, even though the actual might be.

11.g - Formal types are specific, even though the actual can be class-wide.

11.h - The subtype of a formal object of mode **in out** is not static. (This covers the case of AI83-00878.)

11.i - The subtype of a formal parameter of a formal subprogram does not provide an applicable index constraint.

11.j/3 - {*AI05-0239-1*} The profile of a formal subprogram is not subtype conformant with any other profile.

11.k - A generic formal function is not static.

11.l **Ramification:** The exceptions to the above rule about when legality rules are enforced fall into these categories:

11.m - Some rules are checked in the generic declaration, and then again in both the visible and private parts of the instance:

11.n - The parent type of a record extension has to be specific (see 3.9.1). This rule is not checked in the instance body.

11.o - The parent type of a private extension has to be specific (see 7.3). This rule is not checked in the instance body.

11.p/3 - {*AI95-00402-01*} {*AI05-0093-1*} A type with an access discriminant with a default_expression has to be immutably limited. In the generic body, the definition of immutably limited is adjusted in an assume-the-worst manner (thus the rule is checked that way).

11.q - In the declaration of a record extension, if the parent type is nonlimited, then each of the components of the record_extension_part have to be nonlimited (see 3.9.1). In the generic body, this rule is checked in an assume-the-worst manner.

11.r - A preelaborated library unit has to be preelaborable (see 10.2.1). In the generic body, this rule is checked in an assume-the-worst manner.

11.r.1/2 {*AI95-00402-01*} The corrections made by the Corrigendum added a number of such rules, and the Amendment added many more. There doesn't seem to be much value in repeating all of these rules here (as of this writing, there are roughly 33 such rules). As noted below, all such rules are indexed in the AARM.

11.s - For the accessibility rules, the formals have nothing to say about the property in question. Like the above rules, these rules are checked in the generic declaration, and then again in both the visible and private parts of the instance. In the generic body, we have explicit rules that essentially assume the worst (in the cases of type extensions and access-to-subprogram types), and we have runtime checks (in the case of access-to-object types). See 3.9.1, 3.10.2, and 4.6.

11.t We considered runtime checks for access-to-subprogram types as well. However, this would present difficulties for implementations that share generic bodies.

11.u - The rules requiring "reasonable" values for static expressions are ignored when the expected type for the expression is a descendant of a generic formal type other than a generic formal derived type, and do not apply in an instance.

11.v - The rule forbidding two explicit homographs in the same declarative region does not apply in an instance of a generic unit, except that it *does* apply in the declaration of a record extension that appears in the visible part of an instance.

11.w - Some rules do not apply at all in an instance, not even in the visible part:

11.x - Body_stubs are not normally allowed to be multiply nested, but they can be in instances.

11.y Each rule that is an exception is marked with "generic contract issue;" look that up in the index to find them all.

11.z **Ramification:** The Legality Rules are the ones labeled Legality Rules. We are talking about all Legality Rules in the entire language here. Note that, with some exceptions, the legality of a generic unit is checked even if there are no instantiations of the generic unit.

Ramification: {*AI05-0299-1*} {*AI12-0005-1*} The Legality Rules are described here, and the overloading rules were described earlier in this subclause. Presumably, every Static Semantic Item is included indirectly by one of those. Thus, we have covered all the compile-time rules of the language. There is no need to say anything special about the Post-Compilation Rules or the Dynamic Semantic Items. 11.aa/5

Discussion: Here is an example illustrating how this rule is checked: "In the declaration of a record extension, if the parent type is nonlimited, then each of the components of the record_extension_part shall be nonlimited." 11.bb

```
generic                                                                    11.cc
    type Parent is tagged private;
    type Comp is limited private;
package G1 is
    type Extension is new Parent with
        record
            C : Comp; -- Illegal!
        end record;
end G1;
```

The parent type is nonlimited, and the component type is limited, which is illegal. It doesn't matter that one could imagine writing an instantiation with the actual for Comp being nonlimited — we never get to the instance, because the generic itself is illegal. 11.dd/1

On the other hand: 11.ee

```
generic                                                                    11.ff
    type Parent is tagged limited private; -- Parent is limited.
    type Comp is limited private;
package G2 is
    type Extension is new Parent with
        record
            C : Comp; -- OK.
        end record;
end G2;
```

```
type Limited_Tagged is tagged limited null record;                         11.gg
type Non_Limited_Tagged is tagged null record;
```

```
type Limited_Untagged is limited null record;                              11.hh
type Non_Limited_Untagged is null record;
```

```
package Good_1 is new G2(Parent => Limited_Tagged,                         11.ii
                         Comp => Limited_Untagged);
package Good_2 is new G2(Parent => Non_Limited_Tagged,
                         Comp => Non_Limited_Untagged);
package Bad  is new G2(Parent => Non_Limited_Tagged,
                         Comp => Limited_Untagged); -- Illegal!
```

The first instantiation is legal, because in the instance the parent is limited, so the rule is not violated. Likewise, in the second instantiation, the rule is not violated in the instance. However, in the Bad instance, the parent type is nonlimited, and the component type is limited, so this instantiation is illegal. 11.jj

Static Semantics

A generic_instantiation declares an instance; it is equivalent to the instance declaration (a package_-declaration or subprogram_declaration) immediately followed by the instance body, both at the place of the instantiation. 12

Ramification: The declaration and the body of the instance are not "implicit" in the technical sense, even though you can't see them in the program text. Nor are declarations within an instance "implicit" (unless they are implicit by other rules). This is necessary because implicit declarations have special semantics that should not be attached to instances. For a generic subprogram, the profile of a generic_instantiation is that of the instance declaration, by the stated equivalence. 12.a

Ramification: The visible and private parts of a package instance are defined in 7.1, "Package Specifications and Declarations" and 12.7, "Formal Packages". The visible and private parts of a subprogram instance are defined in 8.2, "Scope of Declarations". 12.b

The instance is a copy of the text of the template. [Each use of a formal parameter becomes (in the copy) a use of the actual, as explained below.] An instance of a generic package is a package, that of a generic procedure is a procedure, and that of a generic function is a function. 13

Ramification: An instance is a package or subprogram (because we say so), even though it contains a copy of the generic_formal_part, and therefore doesn't look like one. This is strange, but it's OK, since the syntax rules are overloading rules, and therefore do not apply in an instance. 13.a

13.b **Discussion:** We use a macro-expansion model, with some explicitly-stated exceptions (see below). The main exception is that the interpretation of each construct in a generic unit (especially including the denotation of each name) is determined when the declaration and body of the generic unit (as opposed to the instance) are compiled, and in each instance this interpretation is (a copy of) the template interpretation. In other words, if a construct is interpreted as a name denoting a declaration D, then in an instance, the copy of the construct will still be a name, and will still denote D (or a copy of D). From an implementation point of view, overload resolution is performed on the template, and not on each copy.

13.c We describe the substitution of generic actual parameters by saying (in most cases) that the copy of each generic formal parameter declares a view of the actual. Suppose a name in a generic unit denotes a generic_formal_parameter_declaration. The copy of that name in an instance will denote the copy of that generic_formal_parameter_declaration in the instance. Since the generic_formal_parameter_declaration in the instance declares a view of the actual, the name will denote a view of the actual.

13.d/2 {*AI95-00442-01*} Other properties of the copy (for example, staticness, categories to which types belong) are recalculated for each instance; this is implied by the fact that it's a copy.

13.e/2 {*AI95-00317-01*} Although the generic_formal_part is included in an instance, the declarations in the generic_formal_part are only visible outside the instance in the case of a generic formal package whose formal_package_actual_part includes one or more <> indicators — see 12.7.

14 The interpretation of each construct within a generic declaration or body is determined using the overloading rules when that generic declaration or body is compiled. In an instance, the interpretation of each (copied) construct is the same, except in the case of a name that denotes the generic_declaration or some declaration within the generic unit; the corresponding name in the instance then denotes the corresponding copy of the denoted declaration. The overloading rules do not apply in the instance.

14.a **Ramification:** See 8.6, “The Context of Overload Resolution” for definitions of “interpretation” and “overloading rule”.

14.b Even the generic_formal_parameter_declarations have corresponding declarations in the instance, which declare views of the actuals.

14.c Although the declarations in the instance are copies of those in the generic unit, they often have quite different properties, as explained below. For example a constant declaration in the generic unit might declare a nonstatic constant, whereas the copy of that declaration might declare a static constant. This can happen when the staticness depends on some generic formal.

14.d This rule is partly a ramification of the “current instance” rule in 8.6, “The Context of Overload Resolution”. Note that that rule doesn't cover the generic_formal_part.

14.e/5 {*AI12-0347-1*} Although the overloading rules are not observed in the instance, they are, of course, observed in the generic_instantiation in order to determine the interpretation of the constituents of the generic_instantiation.

14.f Since children are considered to occur within their parent's declarative region, the above rule applies to a name that denotes a child of a generic unit, or a declaration inside such a child.

14.g Since the Syntax Rules are overloading rules, it is possible (legal) to violate them in an instance. For example, it is possible for an instance body to occur in a package_specification, even though the Syntax Rules forbid bodies in package_specifications.

15/5 {*AI12-0407-1*} In an instance, a generic_formal_parameter_declaration declares a view whose properties are identical to those of the actual, except when specified otherwise (in particular, see 6.1.1, “Preconditions and Postconditions”, 12.4, “Formal Objects”, and 12.6, “Formal Subprograms”). Similarly, for a declaration within a generic_formal_parameter_declaration, the corresponding declaration in an instance declares a view whose properties are identical to the corresponding declaration within the declaration of the actual.

15.a **Ramification:** In an instance, there are no “properties” of types and subtypes that come from the formal. The primitive operations of the type come from the formal, but these are declarations in their own right, and are therefore handled separately.

15.b Note that certain properties that come from the actuals are irrelevant in the instance. For example, if an actual type is of a class deeper in the derived-type hierarchy than the formal, it is impossible to call the additional operations of the deeper class in the instance, because any such call would have to be a copy of some corresponding call in the generic unit, which would have been illegal. However, it is sometimes possible to reach into the specification of the instance from outside, and notice such properties. For example, one could pass an object declared in the instance specification to one of the additional operations of the deeper type.

{*AI95-00114-01*} A formal_type_declaration can contain discriminant_specifications, a formal_subprogram_declaration can contain parameter_specifications, and a formal_package_declaration can contain many kinds of declarations. These are all inside the generic unit, and have corresponding declarations in the instance. 15.c/2

This rule implies, for example, that if a subtype in a generic unit is a subtype of a generic formal subtype, then the corresponding subtype in the instance is a subtype of the corresponding actual subtype. 15.d

For a generic_instantiation, if a generic actual is a static [(scalar or string)] subtype, then each use of the corresponding formal parameter within the specification of the instance is considered to be static. (See AI83-00409.) 15.e

Similarly, if a generic actual is a static expression and the corresponding formal parameter has a static [(scalar or string)] subtype, then each use of the formal parameter in the specification of the instance is considered to be static. (See AI83-00505.) 15.f

If a primitive subprogram of a type derived from a generic formal derived tagged type is not overriding (that is, it is a new subprogram), it is possible for the copy of that subprogram in an instance to override a subprogram inherited from the actual. For example: 15.g

```
type T1 is tagged record ... end record;                                                    15.h
{AI12-0005-1} generic                                                                       15.i/5
    type Formal is new T1 with private;
package G is
    type Derived_From_Formal is new Formal with record ... end record;
    procedure Foo(X : in Derived_From_Formal); -- Does not override anything.
end G;
type T2 is new T1 with record ... end record;                                               15.j
procedure Foo(X : in T2);
package Inst is new G(Formal => T2);                                                        15.k
```

In the instance Inst, the declaration of Foo for Derived_From_Formal overrides the Foo inherited from T2. 15.l

Implementation Note: {*8652/0009*} {*AI95-00137-01*} For formal types, an implementation that shares the code among multiple instances of the same generic unit needs to beware that things like parameter passing mechanisms (by-copy vs. by-reference) and aspect_clauses are determined by the actual. 15.m/1

Reason: {*AI12-0407-1*} In particular, we allow differences for aspects that can be specified on generic formal parameters. For instance, Pre (see 6.1.1) can be specified on generic formal subprograms to be added to the Pre of the actual. 15.n/5

[Implicit declarations are also copied, and a name that denotes an implicit declaration in the generic denotes the corresponding copy in the instance. However, for a type declared within the visible part of the generic, a whole new set of primitive subprograms is implicitly declared for use outside the instance, and may differ from the copied set if the properties of the type in some way depend on the properties of some actual type specified in the instantiation. For example, if the type in the generic is derived from a formal private type, then in the instance the type will inherit subprograms from the corresponding actual type. 16

These new implicit declarations occur immediately after the type declaration in the instance, and override the copied ones. The copied ones can be called only from within the instance; the new ones can be called only from outside the instance, although for tagged types, the body of a new one can be executed by a call to an old one.] 17

Proof: This rule is stated officially in 8.3, “Visibility”. 17.a

Ramification: The new ones follow from the class(es) of the formal types. For example, for a type T derived from a generic formal private type, if the actual is Integer, then the copy of T in the instance has a "+" primitive operator, which can be called from outside the instance (assuming T is declared in the visible part of the instance). 17.b

AI83-00398. 17.c

{*AI95-00442-01*} Since an actual type is always in the category determined for the formal, the new subprograms hide all of the copied ones, except for a declaration of "/=" that corresponds to an explicit declaration of "=". Such "/=" operators are special, because unlike other implicit declarations of primitive subprograms, they do not appear by virtue of the class, but because of an explicit declaration of "=". If the declaration of "=" is implicit (and therefore overridden in the instance), then a corresponding implicitly declared "/=" is also overridden. But if the declaration of "=" is explicit (and therefore not overridden in the instance), then a corresponding implicitly declared "/=" is not overridden either, even though it's implicit. 17.d/2

Note that the copied ones can be called from inside the instance, even though they are hidden from all visibility, because the names are resolved in the generic unit — visibility is irrelevant for calls in the instance. 17.e

18 [In the visible part of an instance, an explicit declaration overrides an implicit declaration if they are homographs, as described in 8.3.] On the other hand, an explicit declaration in the private part of an instance overrides an implicit declaration in the instance, only if the corresponding explicit declaration in the generic overrides a corresponding implicit declaration in the generic. Corresponding rules apply to the other kinds of overriding described in 8.3.

18.a **Ramification:** For example:

18.b
```
type Ancestor is tagged null record;
```

18.c
```
generic
    type Formal is new Ancestor with private;
package G is
    type T is new Formal with null record;
    procedure P(X : in T); -- (1)
private
    procedure Q(X : in T); -- (2)
end G;
```

18.d
```
type Actual is new Ancestor with null record;
procedure P(X : in Actual);
procedure Q(X : in Actual);
```

18.e
```
package Instance is new G(Formal => Actual);
```

18.f In the instance, the copy of P at (1) overrides Actual's P, whereas the copy of Q at (2) does not override anything; in implementation terms, it occupies a separate slot in the type descriptor.

18.g/5 **Reason:** {*AI12-0347-1*} The reason for this rule is so a programmer writing a generic_instantiation need not look at the private part of the generic in order to determine which subprograms will be overridden.

Post-Compilation Rules

19 Recursive generic instantiation is not allowed in the following sense: if a given generic unit includes an instantiation of a second generic unit, then the instance generated by this instantiation shall not include an instance of the first generic unit [(whether this instance is generated directly, or indirectly by intermediate instantiations)].

19.a **Discussion:** Note that this rule is not a violation of the generic contract model, because it is not a Legality Rule. Some implementations may be able to check this rule at compile time, but that requires access to all the bodies, so we allow implementations to check the rule at link time.

Dynamic Semantics

20 For the elaboration of a generic_instantiation, each generic_association is first evaluated. If a default is used, an implicit generic_association is assumed for this rule. These evaluations are done in an arbitrary order, except that the evaluation for a default actual takes place after the evaluation for another actual if the default includes a name that denotes the other one. Finally, the instance declaration and body are elaborated.

20.a **Ramification:** Note that if the evaluation of a default depends on some side effect of some other evaluation, the order is still arbitrary.

21 For the evaluation of a generic_association the generic actual parameter is evaluated. Additional actions are performed in the case of a formal object of mode **in** (see 12.4).

21.a **To be honest:** Actually, the actual is evaluated only if evaluation is defined for that kind of construct — we don't actually "evaluate" subtype_marks.

22 NOTE If a formal type is not tagged, then the type is treated as an untagged type within the generic body. Deriving from such a type in a generic body is permitted; the new type does not get a new tag value, even if the actual is tagged. Overriding operations for such a derived type cannot be dispatched to from outside the instance.

22.a **Ramification:** If two overloaded subprograms declared in a generic package specification differ only by the (formal) type of their parameters and results, then there exist legal instantiations for which all calls of these subprograms from outside the instance are ambiguous. For example:

```
generic                                                                    22.b
   type A is (<>);
   type B is private;
package G is
   function Next(X : A) return A;
   function Next(X : B) return B;
end G;

package P is new G(A => Boolean, B => Boolean);                            22.c
-- All calls of P.Next are ambiguous.
```

Ramification: The following example illustrates some of the subtleties of the substitution of formals and actuals: 22.d

```
{AI12-0005-1} generic                                                      22.e/5
    type T1 is private;
    -- A predefined "=" operator is implicitly declared here:
    -- function "="(Left, Right : T1) return Boolean;
    -- Call this "="1.
    ...
package G is
    subtype S1 is T1; -- So we can get our hands on the type from
                      -- outside an instance.
    type T2 is new T1;
    -- An inherited "=" operator is implicitly declared here:
    -- function "="(Left, Right : T2) return Boolean;
    -- Call this "="2.

    T1_Obj : T1 := ...;                                                    22.f
    Bool_1 : Boolean := T1_Obj = T1_Obj;

    T2_Obj : T2 := ...;                                                    22.g
    Bool_2 : Boolean := T2_Obj = T2_Obj;
end G;
...
{AI12-0005-1} package P is                                                 22.h/5
    type My_Int is new Integer;
    -- A predefined "=" operator is implicitly declared here:
    -- function "="(Left, Right : My_Int) return Boolean;
    -- Call this "="3.
    function "="(X, Y : My_Int) return Boolean;
    -- Call this "="4.
    -- "="3 is hidden from all visibility by "="4.
    -- Nonetheless, "="3 can "reemerge" in certain circumstances.
end P;
use P;
...
package I is new G(T1 => My_Int, ...); -- "="5 is declared in I (see below).
use I;

Another_T1_Obj : S1 := 13; -- Can't denote T1, but S1 will do.             22.i
Bool_3 : Boolean := Another_T1_Obj = Another_T1_Obj;

Another_T2_Obj : T2 := 45;                                                 22.j
Bool_4 : Boolean := Another_T2_Obj = Another_T2_Obj;

Double : T2 := T2_Obj + Another_T2_Obj;                                    22.k
```

In the instance I, there is a copy of "="$_1$ (call it "="$_{1i}$) and "="$_2$ (call it "="$_{2i}$). The "="$_{1i}$ and "="$_{2i}$ declare views of the predefined "=" of My_Int (that is, "="$_3$). In the initialization of Bool_1 and Bool_2 in the generic unit G, the names "=" denote "="$_1$ and "="$_2$, respectively. Therefore, the copies of these names in the instances denote "="$_{1i}$ and "="$_{2i}$, respectively. Thus, the initialization of I.Bool_1 and I.Bool_2 call the predefined equality operator of My_Int; they will not call "="$_4$. 22.l

The declarations "="$_{1i}$ and "="$_{2i}$ are hidden from all visibility. This prevents them from being called from outside the instance. 22.m

The declaration of Bool_3 calls "="$_4$. 22.n

The instance I also contains implicit declarations of the primitive operators of T2, such as "=" (call it "="$_5$) and "+". These operations cannot be called from within the instance, but the declaration of Bool_4 calls "="$_5$. 22.o

Examples

23 *Examples of generic instantiations (see 12.1):*

24/5
```
{AI12-0178-1} procedure Swap is new Exchange(Elem => Integer);
procedure Swap is new Exchange(Character);          -- Swap is overloaded
function Square is new Squaring(Integer);   -- "*" of Integer used by default
function Square1 is new Squaring(Item => Matrix, "*" => Matrix_Product);
function Square2 is new Squaring(Matrix, Matrix_Product); -- same as previous
```

25
```
package Int_Vectors is new On_Vectors(Integer, Table, "+");
```

26 *Examples of uses of instantiated units:*

27
```
Swap(A, B);
A := Square(A);
```

28
```
T : Table(1 .. 5) := (10, 20, 30, 40, 50);
N : Integer := Int_Vectors.Sigma(T);  -- 150
                                      -- (see 12.2 for the body of Sigma)
```

29
```
use Int_Vectors;
M : Integer := Sigma(T);  -- 150
```

Inconsistencies With Ada 83

29.a In Ada 83, all explicit actuals are evaluated before all defaults, and the defaults are evaluated in the order of the formal declarations. This ordering requirement is relaxed in Ada 95.

Incompatibilities With Ada 83

29.b We have attempted to remove every violation of the contract model. Any remaining contract model violations should be considered bugs in the RM95. The unfortunate property of reverting to the predefined operators of the actual types is retained for upward compatibility. (Note that fixing this would require subtype conformance rules.) However, tagged types do not revert in this sense.

Extensions to Ada 83

29.c The syntax rule for explicit_generic_actual_parameter is modified to allow a *package_instance_*name.

Wording Changes from Ada 83

29.d The fact that named associations cannot be used for two formal subprograms with the same defining name is moved to AARM-only material, because it is a ramification of other rules, and because it is not of interest to the average user.

29.e/2 {*AI95-00114-01*} The rule that “An explicit explicit_generic_actual_parameter shall not be supplied more than once for a given generic formal parameter” seems to be missing from RM83, although it was clearly the intent.

29.f In the explanation that the instance is a copy of the template, we have left out RM83-12.3(5)'s “apart from the generic formal part”, because it seems that things in the formal part still need to exist in instances. This is particularly true for generic formal packages, where you're sometimes allowed to reach in and denote the formals of the formal package from outside it. This simplifies the explanation of what each name in an instance denotes: there are just two cases: the declaration can be inside or outside (where inside needs to include the generic unit itself). Note that the RM83 approach of listing many cases (see RM83-12.5(5-14)) would have become even more unwieldy with the addition of generic formal packages, and the declarations that occur therein.

29.g We have corrected the definition of the elaboration of a generic_instantiation (RM83-12.3(17)); we don't elaborate entities, and the instance is not “implicit”.

29.h In RM83, there is a rule saying the formal and actual shall match, and then there is much text defining what it means to match. Here, we simply state all the latter text as rules. For example, “A formal foo is matched by an actual greenish bar” becomes “For a formal foo, the actual shall be a greenish bar”. This is necessary to split the Name Resolution Rules from the Legality Rules. Besides, there's really no need to define the concept of matching for generic parameters.

Extensions to Ada 95

29.i/2 {*AI95-00218-03*} An overriding_indicator (see 8.3.1) is allowed on a subprogram instantiation.

Extensions to Ada 2005

29.j/3 {*AI05-0183-1*} An optional aspect_specification can be used in a generic_instantiation. This is described in 13.1.1.

Wording Changes from Ada 2005

{*AI05-0118-1*} **Correction:** Added a definition for positional parameters, as this is missing from Ada 95 and Ada 2005. 29.k/3

Wording Changes from Ada 2012

{*AI12-0205-1*} Added uses of *default_*subtype_marks so formal types can have defaults (see 12.5). 29.l/5

12.4 Formal Objects

[A generic formal object can be used to pass a value or variable to a generic unit.] 1

Language Design Principles

A generic formal object of mode **in** is like a constant initialized to the value of the explicit_generic_actual_parameter. 1.a

A generic formal object of mode **in out** is like a renaming of the explicit_generic_actual_parameter. 1.b

Syntax

{*AI95-00423-01*} {*AI05-0005-1*} {*AI05-0183-1*} formal_object_declaration ::= 2/3
  defining_identifier_list : mode [null_exclusion] subtype_mark [:= default_expression]
    [aspect_specification];
 | defining_identifier_list : mode access_definition [:= default_expression]
    [aspect_specification];

Name Resolution Rules

The expected type for the default_expression, if any, of a formal object is the type of the formal object. 3

For a generic formal object of mode **in**, the expected type for the actual is the type of the formal. 4

{*AI95-00423-01*} For a generic formal object of mode **in out**, the type of the actual shall resolve to the type determined by the subtype_mark, or for a formal_object_declaration with an access_definition, to a specific anonymous access type. If the anonymous access type is an access-to-object type, the type of the actual shall have the same designated type as that of the access_definition. If the anonymous access type is an access-to-subprogram type, the type of the actual shall have a designated profile which is type conformant with that of the access_definition. 5/2

Reason: See the corresponding rule for object_renaming_declarations for a discussion of the reason for this rule. 5.a

Legality Rules

If a generic formal object has a default_expression, then the mode shall be **in** [(either explicitly or by default)]; otherwise, its mode shall be either **in** or **in out**. 6

Ramification: Mode **out** is not allowed for generic formal objects. 6.a

For a generic formal object of mode **in**, the actual shall be an expression. For a generic formal object of mode **in out**, the actual shall be a name that denotes a variable for which renaming is allowed (see 8.5.1). 7

To be honest: The part of this that requires an expression or name is a Name Resolution Rule, but that's too pedantic to worry about. (The part about denoting a variable, and renaming being allowed, is most certainly *not* a Name Resolution Rule.) 7.a

{*AI95-00287-01*} {*AI95-00423-01*} In the case where the type of the formal is defined by an access_definition, the type of the actual and the type of the formal: 8/2

- {*AI95-00423-01*} shall both be access-to-object types with statically matching designated subtypes and with both or neither being access-to-constant types; or 8.1/2

8.2/2 • {*AI95-00423-01*} shall both be access-to-subprogram types with subtype conformant designated profiles.

8.3/5 {*AI95-00423-01*} {*AI12-0287-1*} For a formal_object_declaration of mode **in out** with a null_exclusion or an access_definition that has a null_exclusion, the subtype of the actual matching the formal_object_declaration shall exclude null. In addition, if the actual matching the formal_object_declaration statically denotes the generic formal object of mode **in out** of another generic unit *G*, and the instantiation containing the actual occurs within the body of *G* or within the body of a generic unit declared within the declarative region of *G*, then the declaration of the formal object of *G* shall have a null_exclusion. In addition to the places where Legality Rules normally apply (see 12.3), this rule applies also in the private part of an instance of a generic unit.

8.a/5 **Reason:** {*AI95-00287-01*} {*AI95-00423-01*} {*AI12-0287-1*} This rule prevents "lying". **Null** must never be the value of an object with an explicit null_exclusion. The "in addition" part is an assume-the-worst rule which prevents trouble in generic bodies (including bodies of child units) when the subtype of the formal object of mode **in out** excludes null implicitly. Since a generic formal object of mode **in** is like a constant initialized to the value of the actual, the runtime check performed by the initialization is enough to prevent lying; thus we don't need a Legality Rule for such objects.

Static Semantics

9/2 {*AI95-00255-01*} {*AI95-00423-01*} A formal_object_declaration declares a generic formal object. The default mode is **in**. For a formal object of mode **in**, the nominal subtype is the one denoted by the subtype_mark or access_definition in the declaration of the formal. For a formal object of mode **in out**, its type is determined by the subtype_mark or access_definition in the declaration; its nominal subtype is nonstatic, even if the subtype_mark denotes a static subtype; for a composite type, its nominal subtype is unconstrained if the first subtype of the type is unconstrained[, even if the subtype_mark denotes a constrained subtype].

9.a/2 **Reason:** {*AI95-00255-01*} We require that the subtype is unconstrained because a formal **in out** acts like a renaming, and thus the given subtype is ignored for purposes of matching; any value of the type can be passed. Thus we can assume only that the object is constrained if the first subtype is constrained (and thus there can be no unconstrained subtypes for the type). If we didn't do this, it would be possible to rename or take 'Access of components that could disappear due to an assignment to the whole object.

9.b/2 **Discussion:** {*AI95-00423-01*} The two "even if" clauses are OK even though they don't mention access_definitions; an access subtype can neither be a static subtype nor be a composite type.

10/2 {*AI95-00269-01*} In an instance, a formal_object_declaration of mode **in** is a *full constant declaration* and declares a new stand-alone constant object whose initialization expression is the actual, whereas a formal_object_declaration of mode **in out** declares a view whose properties are identical to those of the actual.

10.a/2 **Ramification:** {*AI95-00287-01*} These rules imply that generic formal objects of mode **in** are passed by copy (or are built-in-place for a limited type), whereas generic formal objects of mode **in out** are passed by reference.

10.b Initialization and finalization happen for the constant declared by a formal_object_declaration of mode **in** as for any constant; see 3.3.1, "Object Declarations" and 7.6, "Assignment and Finalization".

10.c In an instance, the subtype of a generic formal object of mode **in** is as for the equivalent constant. In an instance, the subtype of a generic formal object of mode **in out** is the subtype of the corresponding generic actual.

Dynamic Semantics

11 For the evaluation of a generic_association for a formal object of mode **in**, a constant object is created, the value of the actual parameter is converted to the nominal subtype of the formal object, and assigned to the object[, including any value adjustment — see 7.6].

11.a **Ramification:** This includes evaluating the actual and doing a subtype conversion, which might raise an exception.

11.b **Discussion:** The rule for evaluating a generic_association for a formal object of mode **in out** is covered by the general Dynamic Semantics rule in 12.3.

NOTE The constraints that apply to a generic formal object of mode **in out** are those of the corresponding generic actual parameter (not those implied by the subtype_mark that appears in the formal_object_declaration). Therefore, to avoid confusion, it is recommended that the name of a first subtype be used for the declaration of such a formal object. 12

Ramification: Constraint checks are done at instantiation time for formal objects of mode **in**, but not for formal objects of mode **in out**. 12.a

Extensions to Ada 83

In Ada 83, it is forbidden to pass a (nongeneric) formal parameter of mode **out**, or a subcomponent thereof, to a generic formal object of mode **in out**. This restriction is removed in Ada 95. 12.b

Wording Changes from Ada 83

We make "mode" explicit in the syntax. RM83 refers to the mode without saying what it is. This is also more uniform with the way (nongeneric) formal parameters are defined. 12.c

We considered allowing mode **out** in Ada 95, for uniformity with (nongeneric) formal parameters. The semantics would be identical for modes **in out** and **out**. (Note that generic formal objects of mode **in out** are passed by reference. Note that for (nongeneric) formal parameters that are allowed to be passed by reference, the semantics of **in out** and **out** is the same. The difference might serve as documentation. The same would be true for generic formal objects, if **out** were allowed, so it would be consistent.) We decided not to make this change, because it does not produce any important benefit, and any change has some cost. 12.d

Extensions to Ada 95

{*AI95-00287-01*} A generic formal **in** object can have a limited type. The actual for such an object must be built-in-place via a function_call or aggregate, see 7.5. 12.e/2

{*AI95-00423-01*} A generic formal object can have a null_exclusion or an anonymous access type. 12.f/2

Wording Changes from Ada 95

{*AI95-00255-01*} Clarified that the nominal subtype of a composite formal **in out** object is unconstrained if the first subtype of the type is unconstrained. 12.g/2

{*AI95-00269-01*} Clarified that a formal **in** object can be static when referenced from outside of the instance (by declaring such an object to be a full constant declaration). 12.h/2

Extensions to Ada 2005

{*AI05-0183-1*} An optional aspect_specification can be used in a formal_object_declaration. This is described in 13.1.1. 12.i/3

Extensions to Ada 2012

{*AI12-0287-1*} **Correction:** The Legality Rule for matching null exclusions only applies to formal objects with mode **in out**. (**In** mode formal objects have a run-time check to avoid problems.) This is an extension as some instantiations that were illegal in original Ada 2012 are now legal. 12.j/5

12.5 Formal Types

{*AI95-00442-01*} [A generic formal subtype can be used to pass to a generic unit a subtype whose type is in a certain category of types.] 1/2

Reason: We considered having intermediate syntactic categories formal_integer_type_definition, formal_real_type_definition, and formal_fixed_point_definition, to be more uniform with the syntax rules for non-generic-formal types. However, that would make the rules for formal types slightly more complicated, and it would cause confusion, since formal_discrete_type_definition would not fit into the scheme very well. 1.a

Syntax

{*AI05-0213-1*} formal_type_declaration ::= 2/3
    formal_complete_type_declaration
  | formal_incomplete_type_declaration

{*AI05-0183-1*} {*AI05-0213-1*} {*AI12-0205-1*} formal_complete_type_declaration ::= 2.1/5

type defining_identifier[discriminant_part] **is** formal_type_definition
[**or use** *default*_subtype_mark] [aspect_specification];

2.2/5 {*AI05-0213-1*} {*AI12-0205-1*} formal_incomplete_type_declaration ::=
type defining_identifier[discriminant_part] [**is tagged**]
[**or use** *default*_subtype_mark];

3/2 {*AI95-00251-01*} formal_type_definition ::=
formal_private_type_definition
| formal_derived_type_definition
| formal_discrete_type_definition
| formal_signed_integer_type_definition
| formal_modular_type_definition
| formal_floating_point_definition
| formal_ordinary_fixed_point_definition
| formal_decimal_fixed_point_definition
| formal_array_type_definition
| formal_access_type_definition
| formal_interface_type_definition

Legality Rules

4 For a generic formal subtype, the actual shall be a subtype_mark; it denotes the *(generic) actual subtype*.

4.a **Ramification:** When we say simply "formal" or "actual" (for a generic formal that denotes a subtype) we're talking about the subtype, not the type, since a name that denotes a formal_type_declaration denotes a subtype, and the corresponding actual also denotes a subtype.

Static Semantics

5 A formal_type_declaration declares a *(generic) formal type*, and its first subtype, the *(generic) formal subtype*.

5.a **Ramification:** A subtype (other than the first subtype) of a generic formal type is not a generic formal subtype.

6/3 {*AI95-00442-01*} {*AI05-0213-1*} The form of a formal_type_definition *determines a category (of types)* to which the formal type belongs. For a formal_private_type_definition the reserved words **tagged** and **limited** indicate the category of types (see 12.5.1). The reserved word **tagged** also plays this role in the case of a formal_incomplete_type_declaration. For a formal_derived_type_definition the category of types is the derivation class rooted at the ancestor type. For other formal types, the name of the syntactic category indicates the category of types; a formal_discrete_type_definition defines a discrete type, and so on.

6.a **Reason:** This rule is clearer with the flat syntax rule for formal_type_definition given above. Adding formal_integer_type_definition and others would make this rule harder to state clearly.

6.b/2 {*AI95-00442-01*} We use "category' rather than "class" above, because the requirement that classes are closed under derivation is not important here. Moreover, there are interesting categories that are not closed under derivation. For instance, limited and interface are categories that do not form classes.

Legality Rules

7/2 {*AI95-00442-01*} The actual type shall be in the category determined for the formal.

7.a/2 **Ramification:** {*AI95-00442-01*} For example, if the category determined for the formal is the category of all discrete types, then the actual has to be discrete.

7.b/2 {*AI95-00442-01*} Note that this rule does not require the actual to belong to every category to which the formal belongs. For example, formal private types are in the category of composite types, but the actual need not be composite. Furthermore, one can imagine an infinite number of categories that are just arbitrary sets of types (even though we don't give them names, since they are uninteresting). We don't want this rule to apply to *those* categories.

{*AI95-00114-01*} {*AI95-00442-01*} “Limited” is not an “interesting” category, but “nonlimited” is; it is legal to pass a nonlimited type to a limited formal type, but not the other way around. The reserved word **limited** really represents a category containing both limited and nonlimited types. “Private” is not a category for this purpose; a generic formal private type accepts both private and nonprivate actual types. 7.c/2

{*AI95-00442-01*} It is legal to pass a class-wide subtype as the actual if it is in the right category, so long as the formal has unknown discriminants. 7.d/2

{*AI12-0205-1*} The *default*_subtype_mark, if any, shall denote a subtype which is allowed as an actual subtype for the formal type. 7.1/5

Ramification: {*AI12-0205-1*} This rule is observed at compile time of the generic_declaration, and is consistent with the handling of the default_name of a formal subprogram. This means that a type declared outside of the generic_declaration cannot be used as the *default*_subtype_mark for a formal type that depends on any other formal type. 7.e/5

Static Semantics

{*8652/0037*} {*AI95-00043-01*} {*AI95-00233-01*} {*AI95-00442-01*} {*AI05-0029-1*} {*AI12-0413-1*} [The formal type also belongs to each category that contains the determined category.] The primitive subprograms of the type are as for any type in the determined category. For a formal type other than a formal derived type, these are the predefined operators of the type. For an elementary formal type, the predefined operators are implicitly declared immediately after the declaration of the formal type. For a composite formal type, the predefined operators are implicitly declared either immediately after the declaration of the formal type, or later immediately within the declarative region in which the type is declared according to the rules of 7.3.1. In an instance, the copy of such an implicit declaration declares a view of the predefined operator of the actual type, even if this operator has been overridden for the actual type and even if it is never declared for the actual type, unless the actual type is an untagged record type, in which case it declares a view of the primitive (equality) operator. [The rules specific to formal derived types are given in 12.5.1.] 8/5

Ramification: {*AI95-00442-01*} All properties of the type are as for any type in the category. Some examples: The primitive operations available are as defined by the language for each category. The form of constraint applicable to a formal type in a subtype_indication depends on the category of the type as for a nonformal type. The formal type is tagged if and only if it is declared as a tagged private type, or as a type derived from a (visibly) tagged type. (Note that the actual type might be tagged even if the formal type is not.) 8.a/2

{*AI12-0413-1*} If the primitive equality operator of the (actual) untagged record type is declared abstract, then Program_Error will be raised if the equality operator of the formal type is in fact invoked within an instance of a generic body (see 4.5.2). If the operator is invoked within an instance of the generic spec, the instance is illegal. 8.a.1/5

Reason: {*AI05-0029-1*} The somewhat cryptic phrase “even if it is never declared” is intended to deal with the following oddity: 8.b/3

```
package Q is                                                                    8.c/3
    type T is limited private;
private
    type T is range 1 .. 10;
end Q;

generic                                                                         8.d/3
    type A is array (Positive range <>) of T;
package Q.G is
    A1, A2 : A (1 .. 1);
private
    B : Boolean := A1 = A2;
end Q.G;

with Q.G;                                                                       8.e/3
package R is
   type C is array (Positive range <>) of Q.T;

   package I is new Q.G (C); -- Where is the predefined "=" for C?              8.f/3
end R;
```

An "=" is available for the formal type A in the private part of Q.G. However, no "=" operator is ever declared for type C, because its component type Q.T is limited. Still, in the instance I the name "=" declares a view of the "=" for C which exists-but-is-never-declared. 8.g/3

9 NOTE 1 Generic formal types, like all types, are not named. Instead, a name can denote a generic formal subtype. Within a generic unit, a generic formal type is considered as being distinct from all other (formal or nonformal) types.

9.a **Proof:** This follows from the fact that each formal_type_declaration declares a type.

10 NOTE 2 A discriminant_part is allowed only for certain kinds of types, and therefore only for certain kinds of generic formal types. See 3.7.

10.a/5 **Ramification:** {*AI12-0005-1*} The term "formal floating point type" refers to a type defined by a formal_floating_point_definition. It does not include a formal derived type whose ancestor is a floating point type. Similar terminology applies to the other kinds of formal_type_definition.

Examples

11 *Examples of generic formal types:*

12
```
type Item is private;
type Buffer(Length : Natural) is limited private;
```

13
```
type Enum  is (<>);
type Int   is range <>;
type Angle is delta <>;
type Mass  is digits <>;
```

14
```
type Table is array (Enum) of Item;
```

15 *Example of a generic formal part declaring a formal integer type:*

16
```
generic
   type Rank is range <>;
   First  : Rank := Rank'First;
   Second : Rank := First + 1;  -- the operator "+" of the type Rank
```

Wording Changes from Ada 83

16.a RM83 has separate sections "Generic Formal Xs" and "Matching Rules for Formal Xs" (for various X's) with most of the text redundant between the two. We have combined the two in order to reduce the redundancy. In RM83, there is no "Matching Rules for Formal Types" section; nor is there a "Generic Formal Y Types" section (for Y = Private, Scalar, Array, and Access). This causes, for example, the duplication across all the "Matching Rules for Y Types" sections of the rule that the actual passed to a formal type shall be a subtype; the new organization avoids that problem.

16.b The matching rules are stated more concisely.

16.c We no longer consider the multiplying operators that deliver a result of type *universal_fixed* to be predefined for the various types; there is only one of each in package Standard. Therefore, we need not mention them here as RM83 had to.

Wording Changes from Ada 95

16.d/2 {*8652/0037*} {*AI95-00043-01*} {*AI95-00233-01*} Corrigendum 1 corrected the wording to properly define the location where operators are defined for formal array types. The wording here was inconsistent with that in 7.3.1, "Private Operations". For the Amendment, this wording was corrected again, because it didn't reflect the Corrigendum 1 revisions in 7.3.1.

16.e/2 {*AI95-00251-01*} Formal interface types are defined; see 12.5.5, "Formal Interface Types".

16.f/2 {*AI95-00442-01*} We use "determines a category" rather than class, since not all interesting properties form a class.

Extensions to Ada 2005

16.g/3 {*AI05-0183-1*} An optional aspect_specification can be used in a formal_type_declaration. This is described in 13.1.1.

Wording Changes from Ada 2005

16.h/3 {*AI05-0029-1*} **Correction**: Updated the wording to acknowledge the possibility of operations that are never declared for an actual type but still can be used inside of a generic unit.

16.i/3 {*AI05-0213-1*} {*AI05-0299-1*} Formal incomplete types are added; these are documented as an extension in the next subclause.

Inconsistencies With Ada 2012

{*AI12-0413-1*} **Correction:** Updated the wording to clarify that predefined record equality never reemerges in a generic instantiation. This model was presumed by 4.5.2, but the wording wasn't right for untagged record types with user-defined equality. Therefore, an implementation that strictly implemented the Ada 2012 wording would call the predefined equality for an actual type that is an untagged record type with a user-defined equality, while Ada 2022 implementations would call the primitive (user-defined) equality. This could change the runtime behavior in rare cases. 16.j/5

Extensions to Ada 2012

{*AI12-0205-1*} Generic formal types now can include an optional default subtype_mark. 16.k/5

12.5.1 Formal Private and Derived Types

{*AI95-00442-01*} {*AI05-0213-1*} {*AI12-0445-1*} [In its most general form, the category determined for a formal private type is all types, but the category can be restricted to only nonlimited types or to only tagged types. Similarly, the category for a formal incomplete type is all types but the category can be restricted to only tagged types; unlike other formal types, the actual type can be incompletely defined, and not ready to be frozen (see 13.14). The category determined for a formal derived type is the derivation class rooted at the ancestor type.] 1/5

Proof: {*AI95-00442-01*} {*AI05-0213-1*} The first two rules are given normatively below, and the third rule is given normatively in 12.5; they are repeated here to give a capsule summary of what this subclause is about. 1.a/3

Ramification: {*AI05-0213-1*} Since the actual of a formal incomplete type does not need to be able to be frozen, the actual can be an incomplete type or a partial view before its completion. 1.b/3

Syntax

formal_private_type_definition ::= [[**abstract**] **tagged**] [**limited**] **private** 2

{*AI95-00251-01*} {*AI95-00419-01*} {*AI95-00443-01*} {*AI12-0446-1*} 3/5
formal_derived_type_definition ::=
[**abstract**] [**limited** | **synchronized**] **new** subtype_mark [[**and** interface_list] **with private**]

Legality Rules

If a generic formal type declaration has a known_discriminant_part, then it shall not include a default_expression for a discriminant. 4

Ramification: Consequently, a generic formal subtype with a known_discriminant_part is an indefinite subtype, so the declaration of a stand-alone variable has to provide a constraint on such a subtype, either explicitly, or by its initial value. 4.a

{*AI95-00401-01*} {*AI95-00419-01*} {*AI95-00443-01*} {*AI05-0237-1*} The *ancestor subtype* of a formal derived type is the subtype denoted by the subtype_mark of the formal_derived_type_definition. For a formal derived type declaration, the reserved words **with private** shall appear if and only if the ancestor type is a tagged type; in this case the formal derived type is a private extension of the ancestor type and the ancestor shall not be a class-wide type. [Similarly, an interface_list or the optional reserved words **abstract** or **synchronized** shall appear only if the ancestor type is a tagged type]. The reserved word **limited** or **synchronized** shall appear only if the ancestor type [and any progenitor types] are limited types. The reserved word **synchronized** shall appear (rather than **limited**) if the ancestor type or any of the progenitor types are synchronized interfaces. The ancestor type shall be a limited interface if the reserved word **synchronized** appears. 5/3

Reason: We use the term “ancestor” here instead of “parent” because the actual can be any descendant of the ancestor, not necessarily a direct descendant. 5.a

{*AI95-00419-01*} {*AI05-0005-1*} We require the ancestor type to be limited when **limited** appears so that we avoid oddities like limited integer types. Normally, **limited** means “match anything” for a generic formal, but it was felt that allowing limited elementary types to be declared was just too weird. Integer still matches a formal limited private type; 5.b/3

it is only a problem when the type is known to be elementary. Note that the progenitors are required to be limited by rules in 3.9.4, thus that part of the rule is redundant.

5.c/2 {*AI95-00443-01*} We require that **synchronized** appear if the ancestor or any of the progenitors are synchronized, so that property is explicitly given in the program text – it is not automatically inherited from the ancestors. However, it can be given even if neither the ancestor nor the progenitors are synchronized.

5.1/5 {*AI95-00251-01*} {*AI95-00401-01*} {*AI95-00443-01*} {*AI05-0087-1*} {*AI12-0036-1*} {*AI12-0442-1*} The actual type for a formal derived type shall be a descendant of [the ancestor type and] every progenitor of the formal type. The actual type for a formal derived type shall be tagged if and only if the formal derived type is a private extension. If the reserved word **synchronized** appears in the declaration of the formal derived type, the actual type shall be a synchronized tagged type.

5.d/2 **Proof:** The actual type has to be a descendant of the ancestor type, in order that it be in the correct class. Thus, that part of the rule is redundant.

5.e/3 **Discussion:** {*AI05-0005-1*} For a nonformal private extension, we require the partial view to be synchronized if the full view is synchronized tagged. This does not apply to a formal private extension — it is OK if the formal is not synchronized. Any attempt to extend the formal type will be rechecked in the instance, where the rule disallowing extending a synchronized noninterface type will be enforced. This is consistent with the "no hidden interfaces" rule also applying only to nonformal private extensions, as well as the rule that a limited nonformal private extension implies a limited full type. Formal private extensions are exempted from all these rules to enable the construction of generics that can be used with the widest possible range of types. In particular, an indefinite tagged limited formal private type can match any "concrete" actual tagged type.

5.f/3 {*AI05-0087-1*} A type (including formal types) derived from a limited interface could be nonlimited; we do not want a limited type derived from such an interface to match a nonlimited formal derived type. Otherwise, we could assign limited objects. Thus, we have to explicitly ban this case.

5.g/5 {*AI12-0036-1*} {*AI12-0005-1*} If we allowed actual types whose kind differs from that of the formal derived type, we could allow type conversions that would not be allowed outside of the generic. That would be particularly problematical if the actual is a tagged type with extension components; we could have created an object of the type that is missing those components by converting from the ancestor type to a formal derived type that is not an extension.

6/5 {*AI05-0213-1*} {*AI12-0442-1*} If a formal private or derived subtype is definite, then the actual subtype shall also be definite. If the formal type is nonlimited, the actual type shall be nonlimited.

6.a **Ramification:** On the other hand, for an indefinite formal subtype, the actual can be either definite or indefinite.

6.b/5 **Discussion:** {*AI12-0442-1*} The rule about nonlimited formals applies to both private and derived formal types.

6.1/3 {*AI05-0213-1*} A formal_incomplete_type_declaration declares a formal incomplete type. The only view of a formal incomplete type is an incomplete view. [Thus, a formal incomplete type is subject to the same usage restrictions as any other incomplete type — see 3.10.1.]

7/5 {*AI12-0351-1*} For a generic formal derived type with no discriminant_part, the actual subtype shall be statically compatible with the ancestor subtype. Furthermore:

8/5 - {*AI12-0351-1*} If the ancestor subtype is constrained, the actual subtype shall be constrained;

8.a/5 **Ramification:** {*AI12-0351-1*} In other words, any constraint on the ancestor subtype is considered part of the "contract". Predicates are also considered part of the contract for *any* subtype, via the static compatibility requirement.

9 - If the ancestor subtype is an unconstrained access or composite subtype, the actual subtype shall be unconstrained.

9.a **Reason:** This rule ensures that if a composite constraint is allowed on the formal, one is also allowed on the actual. If the ancestor subtype is an unconstrained scalar subtype, the actual is allowed to be constrained, since a scalar constraint does not cause further constraints to be illegal.

10 - If the ancestor subtype is an unconstrained discriminated subtype, then the actual shall have the same number of discriminants, and each discriminant of the actual shall correspond to a discriminant of the ancestor, in the sense of 3.7.

10.a **Reason:** This ensures that if a discriminant constraint is given on the formal subtype, the corresponding constraint in the instance will make sense, without additional runtime checks. This is not necessary for arrays, since the bounds cannot be overridden in a type extension. An unknown_discriminant_part may be used to relax these matching requirements.

- {*AI95-00231-01*} If the ancestor subtype is an access subtype, the actual subtype shall exclude null if and only if the ancestor subtype excludes null. 10.1/2

  Reason: We require that the "excludes null" property match, because it would be difficult to write a correct generic for a formal access type without knowing this property. Many typical algorithms and techniques will not work for a subtype that excludes null (setting an unused component to **null**, default-initialized objects, and so on). We want this sort of requirement to be reflected in the contract of the generic. 10.b/2

{*AI05-0213-1*} The declaration of a formal derived type shall not have a known_discriminant_part. For a generic formal private or incomplete type with a known_discriminant_part: 11/3

- The actual type shall be a type with the same number of discriminants. 12
- The actual subtype shall be unconstrained. 13
- The subtype of each discriminant of the actual type shall statically match the subtype of the corresponding discriminant of the formal type. 14

  Reason: We considered defining the first and third rule to be called "subtype conformance" for discriminant_parts. We rejected that idea, because it would require implicit (inherited) discriminant_parts, which seemed like too much mechanism. 14.a

{*AI12-0444-1*} [For a generic formal type with an unknown_discriminant_part, the actual may have discriminants, though that is not required, and may be definite or indefinite.] 15/5

{*AI12-0095-1*} When enforcing Legality Rules, for the purposes of determining within a generic body whether a type is unconstrained in any partial view, a discriminated subtype is considered to have a constrained partial view if it is a descendant of an untagged generic formal private or derived type. 15.1/4

Static Semantics

{*AI95-00442-01*} The category determined for a formal private type is as follows: 16/2

Type Definition	*Determined Category* 17/2
limited private	the category of all types
private	the category of all nonlimited types
tagged limited private	the category of all tagged types
tagged private	the category of all nonlimited tagged types

[The presence of the reserved word **abstract** determines whether the actual type may be abstract.] 18

{*AI05-0213-1*} The category determined for a formal incomplete type is the category of all types, unless the formal_type_declaration includes the reserved word **tagged**; in this case, it is the category of all tagged types. 18.1/3

A formal private or derived type is a private or derived type, respectively. A formal derived tagged type is a private extension. [A formal private or derived type is abstract if the reserved word **abstract** appears in its declaration.] 19

{*AI95-00233-01*} {*AI05-0110-1*} For a formal derived type, the characteristics (including components, but excluding discriminants if there is a new discriminant_part), predefined operators, and inherited user-defined primitive subprograms are determined by its ancestor type and its progenitor types (if any), in the same way that those of a derived type are determined by those of its parent type and its progenitor types (see 3.4 and 7.3.1). 20/3

{*8652/0038*} {*AI95-00202*} {*AI95-00233-01*} {*AI95-00401-01*} {*AI05-0029-1*} {*AI05-0110-1*} In an instance, the copy of an implicit declaration of a primitive subprogram of a formal derived type declares a view of the corresponding primitive subprogram of the ancestor or progenitor of the formal derived type, even if this primitive has been overridden for the actual type and even if it is never declared for the actual 21/3

type. When the ancestor or progenitor of the formal derived type is itself a formal type, the copy of the implicit declaration declares a view of the corresponding copied operation of the ancestor or progenitor. [In the case of a formal private extension, however, the tag of the formal type is that of the actual type, so if the tag in a call is statically determined to be that of the formal type, the body executed will be that corresponding to the actual type.]

21.a/3 **Ramification:** {*AI95-00401-01*} {*AI05-0239-1*} The above rule defining the properties of primitive subprograms in an instance applies even if the subprogram has been overridden or hidden for the actual type. This rule is necessary for untagged types, because their primitive subprograms might have been overridden by operations that are not subtype conformant with the operations defined for the class. For tagged types, the rule still applies, but the primitive subprograms will dispatch to the appropriate implementation based on the type and tag of the operands. Even for tagged types, the formal parameter names and default_expressions are determined by those of the primitive subprograms of the specified ancestor type (or progenitor type, for subprograms inherited from an interface type).

21.b/4 **To be honest:** {*AI12-0030-1*} The availability of stream attributes is not formally a characteristic of a type, but it is still determined by the ancestor type for a formal derived type in the same way as the characteristics are. Availability is rechecked in the instance specification.

21.1/5 {*AI12-0419-1*} In an instance, the implicitly composed and additive aspects (see 13.1.1) of a formal type are those of the actual; for a nonoverridable aspect, a formal derived type inherits the aspect if the ancestor or any progenitor has the aspect, according to the rules given in 13.1.

22/1 For a prefix S that denotes a formal indefinite subtype, the following attribute is defined:

23/3 S'Definite {*AI05-0264-1*} S'Definite yields True if the actual subtype corresponding to S is definite; otherwise, it yields False. The value of this attribute is of the predefined type Boolean.

23.a/5 **Discussion:** {*AI95-00114-01*} {*AI12-0005-1*} Whether an actual subtype is definite or indefinite may have a major effect on the algorithm used in a generic. For example, in a generic I/O package, whether to use fixed-length or variable-length records could depend on whether the actual is definite or indefinite. This attribute is essentially a replacement for the Constrained attribute, which is now obsolescent.

Dynamic Semantics

23.1/3 {*AI95-00158-01*} {*AI05-0071-1*} In the case where a formal type has unknown discriminants, and the actual type is a class-wide type *T*'Class:

23.2/2 - {*AI95-00158-01*} For the purposes of defining the primitive operations of the formal type, each of the primitive operations of the actual type is considered to be a subprogram (with an intrinsic calling convention — see 6.3.1) whose body consists of a dispatching call upon the corresponding operation of *T*, with its formal parameters as the actual parameters. If it is a function, the result of the dispatching call is returned.

23.3/2 - {*AI95-00158-01*} If the corresponding operation of *T* has no controlling formal parameters, then the controlling tag value is determined by the context of the call, according to the rules for tag-indeterminate calls (see 3.9.2 and 5.2). In the case where the tag would be statically determined to be that of the formal type, the call raises Program_Error. If such a function is renamed, any call on the renaming raises Program_Error.

23.b/2 **Discussion:** As it states in 6.3.1, the convention of an inherited subprogram of a generic formal tagged type with unknown discriminants is intrinsic.

23.c/2 In the case of a corresponding primitive of T with no controlling formal parameters, the context of the call provides the controlling tag value for the dispatch. If no tag is provided by context, Program_Error is raised rather than resorting to a nondispatching call. For example:

23.d/2
```
generic
   type NT(<>) is new T with private;
    -- Assume T has operation "function Empty return T;"
package G is
   procedure Test(X : in out NT);
end G;
```

```
package body G is                                                          23.e/2
   procedure Test(X : in out NT) is
   begin
      X := Empty;  -- Dispatching based on X'Tag takes
                   -- place if actual is class-wide.
      declare
          Y : NT := Empty;
                   -- If actual is class-wide, this raises Program_Error
                   -- as there is no tag provided by context.
      begin
          X := Y;  -- We never get this far.
      end;
   end Test;
end G;
```

```
type T1 is new T with null record;                                         23.f/2
package I is new G(T1'Class);
```

NOTE 1 The actual type can be abstract only if the formal type is abstract (see 3.9.3). 27

Reason: This is necessary to avoid contract model problems, since one or more of its primitive subprograms are abstract; it is forbidden to create objects of the type, or to declare functions returning the type. 27.a

Ramification: On the other hand, it is OK to pass a nonabstract actual to an abstract formal — **abstract** on the formal indicates that the actual might be abstract. 27.b

NOTE 2 {*AI12-0447-1*} If the formal has a discriminant_part, the actual can be either definite or indefinite. Otherwise, the actual can only be definite. 28/5

Incompatibilities With Ada 83

Ada 83 does not have unknown_discriminant_parts, so it allows indefinite subtypes to be passed to definite formals, and applies a legality rule to the instance body. This is a contract model violation. Ada 95 disallows such cases at the point of the instantiation. The workaround is to add (<>) as the discriminant_part of any formal subtype if it is intended to be used with indefinite actuals. If that's the intent, then there can't be anything in the generic body that would require a definite subtype. 28.a

The check for discriminant subtype matching is changed from a runtime check to a compile-time check. 28.b

Extensions to Ada 95

{*AI95-00251-01*} {*AI95-00401-01*} {*AI95-00419-01*} {*AI95-00443-01*} A generic formal derived type can include progenitors (interfaces) as well as a primary ancestor. It also may include **limited** to indicate that it is a limited type, and **synchronized** to indicate that it is a synchronized type. 28.c/2

Wording Changes from Ada 95

{*8652/0038*} {*AI95-00202-01*} **Corrigendum:** Corrected wording to define the operations that are inherited when the ancestor of a formal type is itself a formal type to avoid anomalies. 28.d/2

{*AI95-00158-01*} Added a semantic description of the meaning of operations of an actual class-wide type, as such a type does not have primitive operations of its own. 28.e/2

{*AI95-00231-01*} Added a matching rule for access subtypes that exclude null. 28.f/2

{*AI95-00233-01*} The wording for the declaration of implicit operations is corrected to be consistent with 7.3.1 as modified by Corrigendum 1. 28.g/2

{*AI95-00442-01*} We change to “determines a category” as that is the new terminology (it avoids confusion, since not all interesting properties form a class). 28.h/2

Incompatibilities With Ada 2005

{*AI05-0087-1*} **Correction:** Added wording to prevent a limited type from being passed to a nonlimited formal derived type. While this was allowed, it would break the contract for the limited type, so hopefully no programs actually depend on that. 28.i/3

Extensions to Ada 2005

{*AI05-0213-1*} Formal incomplete types are a new kind of generic formal; these can be instantiated with incomplete types and unfrozen private types. 28.j/3

Wording Changes from Ada 2005

28.k/3 {*AI05-0029-1*} **Correction:** Updated the wording to acknowledge the possibility of operations that are never declared for an actual type but still can be used inside of a generic unit.

28.l/3 {*AI05-0071-1*} **Correction:** Fixed hole that failed to define what happened for "=" for an untagged private type whose actual is class-wide.

28.m/3 {*AI05-0110-1*} **Correction:** Revised the wording for inheritance of characteristics and operations of formal derived types to be reuse the rules as defined for derived types; this should eliminate holes in the wording which have plagued us since Ada 95 was defined (it has been "corrected" four previous times).

28.n/3 {*AI05-0237-1*} **Correction:** Added missing rule for the ancestors of formal derived types. The added rule would formally be incompatible, but since it would be impossible to instantiate any such generic, this cannot happen outside of test suites and thus is not documented as an incompatibility.

Incompatibilities With Ada 2012

28.o/4 {*AI12-0036-1*} **Corrigendum:** Added a requirement that a tagged type only match a formal derived type that is a private extension. This is necessary to prevent type conversions that would not be allowed outside of the generic. We expect that this will be rare, as it only can happen if the formal derived type does not accurately describe the actual type; in most such cases, extension will be desired and a private extension used so that is allowed.

28.p/5 {*AI12-0351-1*} **Correction:** The predicates of an ancestor subtype are considered part of the contract for a formal derived type, even if the ancestor subtype is unconstrained. This means, for instance, if the ancestor subtype is a subtype of Float with a predicate, then an actual subtype with a different predicate is illegal in Ada 2022 while it would have been allowed in Ada 2012. Cases like this are quite unlikely and will be detected at compile-time if they occur.

Wording Changes from Ada 2012

28.q/4 {*AI12-0095-1*} **Corrigendum:** The assume the worst rule for determining within a generic body whether a type is unconstrained in any partial view was moved here. While AI05-0041-1 added it to 3.10.2, it's also needed (at least) in 4.6 and 6.4.1. Thus, it was moved here so that it applies generally.

12.5.2 Formal Scalar Types

1/2 {*AI95-00442-01*} A *formal scalar type* is one defined by any of the formal_type_definitions in this subclause. [The category determined for a formal scalar type is the category of all discrete, signed integer, modular, floating point, ordinary fixed point, or decimal types.]

1.a/2 **Proof:** {*AI95-00442-01*} The second rule follows from the rule in 12.5 that says that the category is determined by the one given in the name of the syntax production. The effect of the rule is repeated here to give a capsule summary of what this subclause is about.

1.b/2 **Ramification:** {*AI95-00442-01*} The “category of a type” includes any classes that the type belongs to.

Syntax

2 formal_discrete_type_definition ::= (<>)

3 formal_signed_integer_type_definition ::= **range** <>

4 formal_modular_type_definition ::= **mod** <>

5 formal_floating_point_definition ::= **digits** <>

6 formal_ordinary_fixed_point_definition ::= **delta** <>

7 formal_decimal_fixed_point_definition ::= **delta** <> **digits** <>

Legality Rules

8 The actual type for a formal scalar type shall not be a nonstandard numeric type.

8.a **Reason:** This restriction is necessary because nonstandard numeric types have some number of restrictions on their use, which could cause contract model problems in a generic body. Note that nonstandard numeric types can be passed to formal derived and formal private subtypes, assuming they obey all the other rules, and assuming the implementation allows it (being nonstandard means the implementation might disallow anything).

Wording Changes from Ada 95

{*AI95-00442-01*} We change to "determines a category" as that is the new terminology (it avoids confusion, since not all interesting properties form a class). 9.a/2

12.5.3 Formal Array Types

{*AI95-00442-01*} [The category determined for a formal array type is the category of all array types.] 1/2

Proof: {*AI95-00442-01*} This rule follows from the rule in 12.5 that says that the category is determined by the one given in the name of the syntax production. The effect of the rule is repeated here to give a capsule summary of what this subclause is about. 1.a/2

Syntax

formal_array_type_definition ::= array_type_definition 2

Legality Rules

The only form of discrete_subtype_definition that is allowed within the declaration of a generic formal (constrained) array subtype is a subtype_mark. 3

Reason: The reason is the same as for forbidding constraints in subtype_indications (see 12.1). 3.a

For a formal array subtype, the actual subtype shall satisfy the following conditions: 4

- The formal array type and the actual array type shall have the same dimensionality; the formal subtype and the actual subtype shall be either both constrained or both unconstrained. 5
- For each index position, the index types shall be the same, and the index subtypes (if unconstrained), or the index ranges (if constrained), shall statically match (see 4.9.1). 6
- The component subtypes of the formal and actual array types shall statically match. 7
- If the formal type has aliased components, then so shall the actual. 8

Ramification: On the other hand, if the formal's components are not aliased, then the actual's components can be either aliased or not. 8.a

Examples

Example of formal array types: 9

```
-- given the generic package                                                    10
generic                                                                         11
   type Item   is private;
   type Index  is (<>);
   type Vector is array (Index range <>) of Item;
   type Table  is array (Index) of Item;
package P is
   ...
end P;
-- and the types                                                                12
type Mix    is array (Color range <>) of Boolean;                               13
type Option is array (Color) of Boolean;
-- then Mix can match Vector and Option can match Table                         14
package R is new P(Item   => Boolean, Index => Color,                           15
                   Vector => Mix,     Table => Option);
-- Note that Mix cannot match Table and Option cannot match Vector              16
```

Incompatibilities With Ada 83

The check for matching of component subtypes and index subtypes or index ranges is changed from a runtime check to a compile-time check. The Ada 83 rule that "If the component type is not a scalar type, then the component subtypes 16.a

shall be either both constrained or both unconstrained" is removed, since it is subsumed by static matching. Likewise, the rules requiring that component types be the same is subsumed.

Wording Changes from Ada 95

16.b/2 {*AI95-00442-01*} We change to "determines a category" as that is the new terminology (it avoids confusion, since not all interesting properties form a class).

12.5.4 Formal Access Types

1/2 {*AI95-00442-01*} [The category determined for a formal access type is the category of all access types.]

1.a/2 **Proof:** {*AI95-00442-01*} This rule follows from the rule in 12.5 that says that the category is determined by the one given in the name of the syntax production. The effect of the rule is repeated here to give a capsule summary of what this subclause is about.

Syntax

2 formal_access_type_definition ::= access_type_definition

Legality Rules

3 For a formal access-to-object type, the designated subtypes of the formal and actual types shall statically match.

4/2 {*AI95-00231-01*} If and only if the general_access_modifier **constant** applies to the formal, the actual shall be an access-to-constant type. If the general_access_modifier **all** applies to the formal, then the actual shall be a general access-to-variable type (see 3.10). If and only if the formal subtype excludes null, the actual subtype shall exclude null.

4.a **Ramification:** If no _modifier applies to the formal, then the actual type may be either a pool-specific or a general access-to-variable type.

4.a.1/1 **Reason:** {*8652/0109*} {*AI95-00025-01*} Matching an access-to-variable to a formal access-to-constant type cannot be allowed. If it were allowed, it would be possible to create an access-to-variable value designating a constant.

4.b/2 {*AI95-00231-01*} We require that the "excludes null" property match, because it would be difficult to write a correct generic for a formal access type without knowing this property. Many typical algorithms and techniques will not work for a subtype that excludes null (setting an unused component to **null**, default-initialized objects, and so on). Even Ada.Unchecked_Deallocation would fail for a subtype that excludes null. Most generics would end up with comments saying that they are not intended to work for subtypes that exclude null. We would rather that this sort of requirement be reflected in the contract of the generic.

5/3 {*AI05-0239-1*} {*AI05-0288-1*} For a formal access-to-subprogram subtype, the designated profiles of the formal and the actual shall be subtype conformant.

Examples

6 *Example of formal access types:*

7
```
-- the formal types of the generic package
```

8
```
generic
   type Node is private;
   type Link is access Node;
package P is
   ...
end P;
```

9
```
-- can be matched by the actual types
```

10
```
type Car;
type Car_Name is access Car;
```

```
type Car is                                                                    11
   record
      Pred, Succ : Car_Name;
      Number     : License_Number;
      Owner      : Person;
   end record;
-- in the following generic instantiation                                      12
package R is new P(Node => Car, Link => Car_Name);                             13
```

Incompatibilities With Ada 83

The check for matching of designated subtypes is changed from a runtime check to a compile-time check. The Ada 83 rule that "If the designated type is other than a scalar type, then the designated subtypes shall be either both constrained or both unconstrained" is removed, since it is subsumed by static matching. 13.a

Extensions to Ada 83

Formal access-to-subprogram subtypes and formal general access types are new concepts. 13.b

Wording Changes from Ada 95

{*AI95-00231-01*} Added a matching rule for subtypes that exclude null. 13.c/2

{*AI95-00442-01*} We change to "determines a category" as that is the new terminology (it avoids confusion, since not all interesting properties form a class). 13.d/2

Incompatibilities With Ada 2005

{*AI05-0288-1*} **Correction:** Matching of formal access-to-subprogram types now uses subtype conformance rather than mode conformance, which is needed to plug a hole. This could cause some instantiations legal in Ada 95 and Ada 2005 to be rejected in Ada 2012. We believe that formal access-to-subprogram types occur rarely, and actuals that are not subtype conformant are rarer still, so this should not happen often. (In addition, one popular compiler has a bug that causes such instances to be rejected, so no code compiled with that compiler could have an incompatibility.) 13.e/3

12.5.5 Formal Interface Types

{*AI95-00251-01*} {*AI95-00442-01*} [The category determined for a formal interface type is the category of all interface types.] 1/2

Proof: {*AI95-00442-01*} This rule follows from the rule in 12.5 that says that the category is determined by the one given in the name of the syntax production. The effect of the rule is repeated here to give a capsule summary of what this subclause is about. 1.a/2

Ramification: Here we're taking advantage of our switch in terminology from "determined class" to "determined category"; by saying "category" rather than "class", we require that any actual type be an interface type, not just some type derived from an interface type. 1.b/2

Syntax

{*AI95-00251-01*} formal_interface_type_definition ::= interface_type_definition 2/2

Legality Rules

{*AI95-00251*} {*AI95-00401*} The actual type shall be a descendant of every progenitor of the formal type. 3/2

{*AI95-00345*} The actual type shall be a limited, task, protected, or synchronized interface if and only if the formal type is also, respectively, a limited, task, protected, or synchronized interface. 4/2

Discussion: We require the kind of interface type to match exactly because without that it is almost impossible to properly implement the interface. 4.a/2

Examples

5/5 *{AI12-0429-1} {AI12-0442-1} Example of the use of a generic with a formal interface type, to establish a standard interface that all tasks will implement so they can be managed appropriately by an application-specific scheduler:*

6/2
```
{AI95-00433-01} type Root_Work_Item is tagged private;
```

7/2
```
{AI95-00433-01} generic
   type Managed_Task is task interface;
   type Work_Item(<>) is new Root_Work_Item with private;
package Server_Manager is
   task type Server is new Managed_Task with
      entry Start(Data : in out Work_Item);
   end Server;
end Server_Manager;
```

Extensions to Ada 95

8.a/2 *{AI95-00251-01} {AI95-00345-01} {AI95-00401-01} {AI95-00442-01}* The formal interface type is new.

12.6 Formal Subprograms

1 [Formal subprograms can be used to pass callable entities to a generic unit.]

Language Design Principles

1.a Generic formal subprograms are like renames of the explicit_generic_actual_parameter.

Syntax

2/2 *{AI95-00260-02}* formal_subprogram_declaration ::= formal_concrete_subprogram_declaration
| formal_abstract_subprogram_declaration

2.1/3 *{AI95-00260-02} {AI05-0183-1}* formal_concrete_subprogram_declaration ::=
with subprogram_specification [**is** subprogram_default]
[aspect_specification];

2.2/3 *{AI95-00260-02} {AI05-0183-1}* formal_abstract_subprogram_declaration ::=
with subprogram_specification **is abstract** [subprogram_default]
[aspect_specification];

3/2 *{AI95-00348-01}* subprogram_default ::= default_name | <> | **null**

4 default_name ::= name

4.1/2 *{AI95-00260-02} {AI95-00348-01}* A subprogram_default of **null** shall not be specified for a formal function or for a formal_abstract_subprogram_declaration.

4.a/2 **Reason:** There are no null functions because the return value has to be constructed somehow. We don't allow null for abstract formal procedures, as the operation is dispatching. It doesn't seem appropriate (or useful) to say that the implementation of something is null in the formal type and all possible descendants of that type. This also would define a dispatching operation that doesn't correspond to a slot in the tag of the controlling type, which would be a new concept. Finally, additional rules would be needed to define the meaning of a dispatching null procedure (for instance, the convention of such a subprogram should be intrinsic, but that's not what the language says). It doesn't seem worth the effort.

Name Resolution Rules

5 The expected profile for the default_name, if any, is that of the formal subprogram.

5.a/3 **Ramification:** *{AI05-0299-1}* This rule, unlike others in this subclause, is observed at compile time of the generic_declaration.

The evaluation of the default_name takes place during the elaboration of each instantiation that uses the default, as defined in 12.3, "Generic Instantiation". 5.b

For a generic formal subprogram, the expected profile for the actual is that of the formal subprogram. 6

Legality Rules

{*AI05-0239-1*} The profiles of the formal and any named default shall be mode conformant. 7/3

Ramification: {*AI05-0299-1*} This rule, unlike others in this subclause, is checked at compile time of the generic_declaration. 7.a/3

{*AI05-0239-1*} The profiles of the formal and actual shall be mode conformant. 8/3

{*AI95-00423-01*} For a parameter or result subtype of a formal_subprogram_declaration that has an explicit null_exclusion: 8.1/2

- {*AI12-0183-1*} {*AI12-0287-1*} if the actual matching the formal_subprogram_declaration statically denotes a generic formal subprogram of another generic unit *G*, and the instantiation containing the actual occurs within the body of a generic unit *G* or within the body of a generic unit declared within the declarative region of the generic unit *G*, then the corresponding parameter or result type of the formal subprogram of *G* shall have a null_exclusion; 8.2/5
- otherwise, the subtype of the corresponding parameter or result type of the actual matching the formal_subprogram_declaration shall exclude null. In addition to the places where Legality Rules normally apply (see 12.3), this rule applies also in the private part of an instance of a generic unit. 8.3/2

Reason: This rule prevents "lying". **Null** must never be the value of a parameter or result with an explicit null_exclusion. The first bullet is an assume-the-worst rule which prevents trouble in generic bodies (including bodies of child generics) when the formal subtype excludes null implicitly. 8.a/2

{*AI12-0204-1*} If the named default, if any, is a prefixed view, the prefix of that view shall denote an object for which renaming is allowed (see 8.5.1). Similarly, if the actual subprogram in an instantiation is a prefixed view, the prefix of that view shall denote an object for which renaming is allowed. 8.4/5

Reason: {*AI12-0204-1*} The prefix in such a case is essentially renamed at the point of the instantiation and passed to any calls of the formal subprogram in the generic. If the prefix isn't legal to rename, that doesn't make sense (and allowing it might end up passing a nonexistent object to some calls). 8.b/5

{*AI95-00260-02*} {*AI05-0296-1*} If a formal parameter of a formal_abstract_subprogram_declaration is of a specific tagged type *T* or of an anonymous access type designating a specific tagged type *T*, *T* is called a *controlling type* of the formal_abstract_subprogram_declaration. Similarly, if the result of a formal_abstract_subprogram_declaration for a function is of a specific tagged type *T* or of an anonymous access type designating a specific tagged type *T*, *T* is called a controlling type of the formal_abstract_subprogram_declaration. A formal_abstract_subprogram_declaration shall have exactly one controlling type, and that type shall not be incomplete. 8.5/5

Ramification: The specific tagged type could be any of a formal tagged private type, a formal derived type, a formal interface type, or a normal tagged type. While the last case doesn't seem to be very useful, there isn't any good reason for disallowing it. This rule ensures that the operation is a dispatching operation of some type, and that we unambiguously know what that type is. 8.c/2

We informally call a subprogram declared by a formal_abstract_subprogram_declaration an *abstract formal subprogram*, but we do not use this term in normative wording. (We do use it often in these notes.) 8.d/2

{*AI95-00260-02*} {*AI12-0165-1*} The actual subprogram for a formal_abstract_subprogram_declaration shall be: 8.6/5

- a dispatching operation of the controlling type; or 8.7/5
- if the controlling type is a formal type, and the actual type corresponding to that formal type is a specific type *T*, a dispatching operation of type *T*; or 8.8/5

8.9/5 • if the controlling type is a formal type, and the actual type is a class-wide type *T*'Class, an implicitly declared subprogram corresponding to a primitive operation of type *T* (see Static Semantics below).

8.e/2 **To be honest:** We mean the controlling type of the formal_abstract_subprogram_declaration, of course. Saying that gets unwieldy and redundant (so says at least one reviewer, anyway).

8.e.1/5 **Ramification:** {*AI12-0165-1*} This means that the actual is a primitive operation of the controlling type, an abstract formal subprogram, or the implicitly available primitive operation for a class-wide type.

8.e.2/5 {*AI12-0165-1*} {*AI12-0005-1*} An explicit class-wide operation cannot be used for the actual of a formal abstract subprogram. Such an operation is never primitive (only specific types have primitive subprograms).

8.f/5 {*AI12-0165-1*} This could happen in a case like:

8.g/2
```
generic
   type T(<>) is tagged private;
   with procedure Foo (Obj : in T) is abstract;
package P ...
```

8.h/2
```
package New_P is new P (Something'Class, Some_Proc);
```

8.i/5 {*AI12-0165-1*} If Some_Proc is an explicit class-wide operation, the instantiation here is illegal, because Some_Proc is not a primitive operation of Something'Class (there are no such operations).

8.i.1/5 {*AI12-0165-1*} However, if Some_Proc is a primitive operation of type Something, then the instantiation is legal; the actual is the implicitly generated subprogram described in Static Semantics below. This is not a problem, since the rules given in 12.5.1 explain how this routine dispatches even though its parameter is class-wide.

8.i.2/5 {*AI12-0165-1*} We allow this special case because it is possible for a class-wide operation to be primitive inside of an instance, and the contract model does not allow us to make such cases illegal. As such, it seems inconsistent to not allow the same in explicit instantiations.

8.j/5 {*AI12-0005-1*} {*AI12-0165-1*} Specifically, since it is possible for a formal tagged type to be instantiated with a class-wide type, it is possible for the (real) controlling type to be class-wide as in the following unusual case:

8.k/5
```
{AI12-0005-1} generic
   type NT(<>) is new T with private;
   -- Presume that T has the following primitive operation:
   -- procedure Bar (Obj : in T);
package Gr ...
```

8.l/2
```
package body Gr is
   package New_P2 is new P (NT, Foo => Bar);
end Gr;
```

8.m/2
```
package New_Gr is new Gr (Something'Class);
```

8.n/5 {*AI12-0165-1*} The instantiation of New_P2 is legal, since Bar is a dispatching operation of the actual type of the controlling type of the abstract formal subprogram Foo. Again, the rules given in 12.5.1 explain how this routine dispatches.

8.o/2 Note that this legality rule never needs to be rechecked in an instance (that contains a nested instantiation). The rule only talks about the actual type of the instantiation; it does not require looking further; if the actual type is in fact a formal type, we do not intend looking at the actual for that formal.

Static Semantics

9 A formal_subprogram_declaration declares a generic formal subprogram. The types of the formal parameters and result, if any, of the formal subprogram are those determined by the subtype_marks given in the formal_subprogram_declaration; however, independent of the particular subtypes that are denoted by the subtype_marks, the nominal subtypes of the formal parameters and result, if any, are defined to be nonstatic, and unconstrained if of an array type [(no applicable index constraint is provided in a call on a formal subprogram)]. In an instance, a formal_subprogram_declaration declares a view of the actual. The profile of this view takes its subtypes and calling convention from the original profile of the actual entity, while taking the formal parameter names and default_expressions from the profile given in the formal_subprogram_declaration. The view is a function or procedure, never an entry.

9.a **Discussion:** This rule is intended to be the same as the one for renamings-as-declarations, where the formal_subprogram_declaration is analogous to a renaming-as-declaration, and the actual is analogous to the renamed view.

{*AI05-0071-1*} {*AI05-0131-1*} If a subtype_mark in the profile of the formal_subprogram_declaration denotes a formal private or formal derived type and the actual type for this formal type is a class-wide type *T*'Class, then for the purposes of resolving the corresponding actual subprogram at the point of the instantiation, certain implicit declarations may be available as possible resolutions as follows: 9.1/3

For each primitive subprogram of *T* that is directly visible at the point of the instantiation, and that has at least one controlling formal parameter, a corresponding implicitly declared subprogram with the same defining name, and having the same profile as the primitive subprogram except that *T* is systematically replaced by *T*'Class in the types of its profile, is potentially use-visible. The body of such a subprogram is as defined in 12.5.1 for primitive subprograms of a formal type when the actual type is class-wide. 9.2/3

Reason: {*AI05-0071-1*} {*AI05-0131-1*} This gives the same capabilities to formal subprograms as those that primitive operations of the formal type have when the actual type is class-wide. We do not want to discourage the use of explicit declarations for (formal) subprograms! 9.b/3

Implementation Note: {*AI05-0071-1*} {*AI05-0131-1*} Although the above wording seems to require constructing implicit versions of all of the primitive subprograms of type *T*, it should be clear that a compiler only needs to consider those that could possibly resolve to the corresponding actual subprogram. For instance, if the formal subprogram is a procedure with two parameters, and the actual subprogram name is Bar (either given explicitly or by default), the compiler need not consider primitives that are functions, that have the wrong number of parameters, that have defining names other than Bar, and so on; thus it does not need to construct implicit declarations for those primitives. 9.c/3

Ramification: {*AI05-0071-1*} {*AI05-0131-1*} Functions that only have a controlling result and do not have a controlling parameter of *T* are not covered by this rule, as any call would be required to raise Program_Error by 12.5.1. It is better to detect the error earlier than at run time. 9.d/3

If a generic unit has a subprogram_default specified by a box, and the corresponding actual parameter is omitted, then it is equivalent to an explicit actual parameter that is a usage name identical to the defining name of the formal. 10

{*AI95-00348-01*} If a generic unit has a subprogram_default specified by the reserved word **null**, and the corresponding actual parameter is omitted, then it is equivalent to an explicit actual parameter that is a null procedure having the profile given in the formal_subprogram_declaration. 10.1/2

{*AI95-00260-02*} The subprogram declared by a formal_abstract_subprogram_declaration with a controlling type *T* is a dispatching operation of type *T*. 10.2/2

Reason: This is necessary to trigger all of the dispatching operation rules. It otherwise would not be considered a dispatching operation, as formal subprograms are never primitive operations. 10.a/2

NOTE 1 {*AI12-0442-1*} The matching rules for formal subprograms state requirements that are similar to those applying to subprogram_renaming_declarations (see 8.5.4). In particular, the name of a parameter of the formal subprogram can be different from that of the corresponding parameter of the actual subprogram; similarly, for these parameters, default_expressions can be different. 11/5

NOTE 2 The constraints that apply to a parameter of a formal subprogram are those of the corresponding formal parameter of the matching actual subprogram (not those implied by the corresponding subtype_mark in the _specification of the formal subprogram). A similar remark applies to the result of a function. Therefore, to avoid confusion, it is recommended that the name of a first subtype be used in any declaration of a formal subprogram. 12

NOTE 3 The subtype specified for a formal parameter of a generic formal subprogram can be any visible subtype, including a generic formal subtype of the same generic_formal_part. 13

NOTE 4 A formal subprogram is matched by an attribute of a type if the attribute is a function with a matching specification. An enumeration literal of a given type matches a parameterless formal function whose result type is the given type. 14

NOTE 5 {*AI12-0347-1*} A default_name denotes an entity that is visible or directly visible at the place of the generic_declaration; a box used as a default is equivalent to a name that denotes an entity that is directly visible at the place of the generic_instantiation. 15/5

Proof: Visibility and name resolution are applied to the equivalent explicit actual parameter. 15.a

NOTE 6 {*AI95-00260-02*} The actual subprogram cannot be abstract unless the formal subprogram is a formal_abstract_subprogram_declaration (see 3.9.3). 16/2

16.1/5 NOTE 7 {*AI95-00260-02*} {*AI12-0442-1*} The subprogram declared by a formal_abstract_subprogram_declaration is an abstract subprogram. All calls on a subprogram declared by a formal_abstract_subprogram_declaration are limited to dispatching calls. See 3.9.3.

16.2/2 NOTE 8 {*AI95-00348-01*} A null procedure as a subprogram default has convention Intrinsic (see 6.3.1).

16.a.1/2 **Proof:** This is an implicitly declared subprogram, so it has convention Intrinsic as defined in 6.3.1.

Examples

17 *Examples of generic formal subprograms:*

18/2
```
{AI95-00433-01} with function "+"(X, Y : Item) return Item is <>;
with function Image(X : Enum) return String is Enum'Image;
with procedure Update is Default_Update;
with procedure Pre_Action(X : in Item) is null;  -- defaults to no action
with procedure Write(S    : not null access Root_Stream_Type'Class;
                     Desc : Descriptor)
                     is abstract Descriptor'Write;  -- see 13.13.2
-- Dispatching operation on Descriptor with default
```

19
```
-- given the generic procedure declaration
```

20
```
generic
   with procedure Action (X : in Item);
procedure Iterate(Seq : in Item_Sequence);
```

21
```
-- and the procedure
```

22
```
procedure Put_Item(X : in Item);
```

23
```
-- the following instantiation is possible
```

24
```
procedure Put_List is new Iterate(Action => Put_Item);
```

Extensions to Ada 95

24.a/2 {*AI95-00260-02*} The formal_abstract_subprogram_declaration is new. It allows the passing of dispatching operations to generic units.

24.b/2 {*AI95-00348-01*} The formal subprogram default of **null** is new. It allows the default of a generic procedure to do nothing, such as for passing a debugging routine.

Wording Changes from Ada 95

24.c/2 {*AI95-00423-01*} Added matching rules for null_exclusions.

Incompatibilities With Ada 2005

24.d/3 {*AI05-0296-1*} It is now illegal to declare a formal abstract subprogram whose controlling type is incomplete. It was never intended to allow that, and such a type would have to come from outside of the generic unit in Ada 2005, so it is unlikely to be useful. Moreover, a dispatching call on the subprogram is likely to fail in many implementations. So it is very unlikely that any code will need to be changed because of this new rule.

Extensions to Ada 2005

24.e/3 {*AI05-0071-1*} {*AI05-0131-1*} **Correction:** Added construction of implicit subprograms for primitives of class-wide actual types, to make it possible to import subprograms via formal subprograms as well as by implicit primitive operations of a formal type. (This is a **Correction** as it is very important for the usability of indefinite containers when instantiated with class-wide types; thus we want Ada 2005 implementations to support it.)

24.f/3 {*AI05-0183-1*} An optional aspect_specification can be used in a formal_concrete_subprogram_declaration and a formal_abstract_subprogram_declaration. This is described in 13.1.1.

Incompatibilities With Ada 2012

24.g/5 {*AI12-0204-1*} **Correction:** Added a rule to ensure that for an actual subprogram that is a prefixed view, the prefix continues to exist during the life of the instantiation. If the prefix is a subcomponent that depends on discriminants, Ada 2005 and 2012 would have allowed the prefix while Ada 2022 would not. Without this change, explicit forms (renaming the object and then using that in instantiations) would be safer than directly using the prefixed view; that's inconsistent with other kinds of actual subprograms.

Extensions to Ada 2012

{*AI12-0165-1*} **Correction:** We now allow the actual for a formal_abstract_subprogram_declaration to be an implicitly declared subprogram for a class-wide type. The rules already required a compiler to be able to construct this subprogram for use in an instantiation that occurs inside of a generic unit, so it made no sense to prevent it from being used explicitly as well (in the same way that the previous extension allows them to be used for other kinds of formal subprograms). 24.h/5

Wording Changes from Ada 2012

{*AI12-0287-1*} **Correction:** Added wording to ensure that the object subject to a Legality Rule can be determined at compile-time. The alternative being nonsense, we treat this as a wording change. 24.i/5

12.7 Formal Packages

[Formal packages can be used to pass packages to a generic unit. The formal_package_declaration declares that the formal package is an instance of a given generic package. Upon instantiation, the actual package has to be an instance of that generic package.] 1

Syntax

{*AI05-0183-1*} formal_package_declaration ::= 2/3
  with package defining_identifier **is new** *generic_package_*name formal_package_actual_part
    [aspect_specification];

{*AI95-00317-01*} formal_package_actual_part ::= 3/2
  ([**others** =>] <>)
 | [generic_actual_part]
 | (formal_package_association {, formal_package_association} [, **others** => <>])

{*AI95-00317-01*} formal_package_association ::= 3.1/2
  generic_association
 | *generic_formal_parameter_*selector_name => <>

{*AI95-00317-01*} Any positional formal_package_associations shall precede any named formal_-package_associations. 3.2/2

Legality Rules

The *generic_package_*name shall denote a generic package (the *template* for the formal package); the formal package is an instance of the template. 4

{*AI05-0025-1*} The *generic_formal_parameter_*selector_name of a formal_package_association shall denote a generic_formal_parameter_declaration of the template. If two or more formal subprograms of the template have the same defining name, then named associations are not allowed for the corresponding actuals. 4.1/3

{*AI95-00398-01*} A formal_package_actual_part shall contain at most one formal_package_association for each formal parameter. If the formal_package_actual_part does not include "**others** => <>", each formal parameter without an association shall have a default_expression or subprogram_default. 4.2/3

{*AI05-0200-1*} The rules for matching between formal_package_associations and the generic formals of the template are as follows: 4.3/3

- If all of the formal_package_associations are given by generic associations, the explicit_generic_actual_parameters of the formal_package_associations shall be legal for an instantiation of the template. 4.4/3

4.5/5 • {*AI05-0200-1*} If a formal_package_association for a formal type *T* of the template is given by <>, then the formal_package_association for any other generic_formal_parameter_declaration of the template that mentions *T* directly or indirectly shall also be given by <>.

4.a/3 **Discussion:** {*AI05-0200-1*} The above rule is simple to state, though it does not reflect the fact that the formal package functions like an instantiation of a special kind, where each box association for a generic_formal_parameter_declaration *F* is replaced with a new entity *F'* that has the same characteristics as *F*: if *F* is a formal discrete type then *F'* is a discrete type, if *F* is a formal subprogram then *F'* is a subprogram with a similar signature, etc. In practice this is achieved by making the association into a copy of the declaration of the generic formal.

5/2 {*AI95-00317-01*} The actual shall be an instance of the template. If the formal_package_actual_part is (<>) or (**others** => <>), [then the actual may be any instance of the template]; otherwise, certain of the actual parameters of the actual instance shall match the corresponding actual parameters of the formal package, determined as follows:

5.1/2 • {*AI95-00317-01*} If the formal_package_actual_part includes generic_associations as well as associations with <>, then only the actual parameters specified explicitly with generic_associations are required to match;

5.2/2 • {*AI95-00317-01*} Otherwise, all actual parameters shall match[, whether any actual parameter is given explicitly or by default].

5.3/2 {*AI95-00317-01*} The rules for matching of actual parameters between the actual instance and the formal package are as follows:

6/2 • {*AI95-00317-01*} For a formal object of mode **in**, the actuals match if they are static expressions with the same value, or if they statically denote the same constant, or if they are both the literal **null**.

6.a **Reason:** We can't simply require full conformance between the two actual parameter expressions, because the two expressions are being evaluated at different times.

7 • For a formal subtype, the actuals match if they denote statically matching subtypes.

8 • For other kinds of formals, the actuals match if they statically denote the same entity.

8.1/1 {*8652/0039*} {*AI95-00213-01*} For the purposes of matching, any actual parameter that is the name of a formal object of mode **in** is replaced by the formal object's actual expression (recursively).

Static Semantics

9 A formal_package_declaration declares a generic formal package.

10/2 {*AI95-00317-01*} The visible part of a formal package includes the first list of basic_declarative_items of the package_specification. In addition, for each actual parameter that is not required to match, a copy of the declaration of the corresponding formal parameter of the template is included in the visible part of the formal package. If the copied declaration is for a formal type, copies of the implicit declarations of the primitive subprograms of the formal type are also included in the visible part of the formal package.

10.a/2 **Ramification:** {*AI95-00317-01*} If the formal_package_actual_part is (<>), then the declarations that occur immediately within the generic_formal_part of the template for the formal package are visible outside the formal package, and can be denoted by expanded names outside the formal package. If only some of the actual parameters are given by <>, then the declaration corresponding to those parameters (but not the others) are made visible.

10.b/3 **Reason:** {*AI05-0005-1*} We always want either the actuals or the formals of an instance to be nameable from outside, but never both. If both were nameable, one would get some funny anomalies since they denote the same entity, but, in the case of types at least, they might have different and inconsistent sets of primitive operators due to predefined operator "reemergence", Formal derived types exacerbate the difference. We want the implicit declarations of the generic_formal_part as well as the explicit declarations, so we get operations on the formal types.

10.c **Ramification:** A generic formal package is a package, and is an instance. Hence, it is possible to pass a generic formal package as an actual to another generic formal package.

{*AI95-00317-01*} For the purposes of matching, if the actual instance *A* is itself a formal package, then the actual parameters of *A* are those specified explicitly or implicitly in the formal_package_actual_part for *A*, plus, for those not specified, the copies of the formal parameters of the template included in the visible part of *A*. 11/2

Examples

{*AI95-00433-01*} *Example of a generic package with formal package parameters:* 12/2

```
with Ada.Containers.Ordered_Maps;  -- see A.18.6                                13/2
generic
   with package Mapping_1 is new Ada.Containers.Ordered_Maps(<>);
   with package Mapping_2 is new Ada.Containers.Ordered_Maps
                                    (Key_Type => Mapping_1.Element_Type,
                                     others => <>);
package Ordered_Join is
   -- Provide a "join" between two mappings

   subtype Key_Type is Mapping_1.Key_Type;                                      14/2
   subtype Element_Type is Mapping_2.Element_Type;

   function Lookup(Key : Key_Type) return Element_Type;                         15/2

   ...                                                                          16/2
end Ordered_Join;
```

{*AI95-00433-01*} *Example of an instantiation of a package with formal packages:* 17/2

```
with Ada.Containers.Ordered_Maps;                                               18/2
package Symbol_Package is

{AI12-0178-1}    subtype Key_String is String(1..5);                            19/5
   type String_Id is ...

   type Symbol_Info is ...                                                      20/2

{AI12-0178-1}    package String_Table is new Ada.Containers.Ordered_Maps        21/5
           (Key_Type => Key_String,
            Element_Type => String_Id);

   package Symbol_Table is new Ada.Containers.Ordered_Maps                      22/2
           (Key_Type => String_Id,
            Element_Type => Symbol_Info);

   package String_Info is new Ordered_Join(Mapping_1 => String_Table,           23/2
                                           Mapping_2 => Symbol_Table);

   Apple_Info : constant Symbol_Info := String_Info.Lookup("Apple");            24/2

end Symbol_Package;                                                             25/2
```

Extensions to Ada 83

Formal packages are new to Ada 95. 25.a

Extensions to Ada 95

{*AI95-00317-01*} {*AI95-00398-01*} It's now allowed to mix actuals of a formal package that are specified with those that are not specified. 25.b/2

Wording Changes from Ada 95

{*8652/0039*} {*AI95-00213-01*} **Corrigendum:** Corrected the description of formal package matching to say that formal parameters are always replaced by their actual parameters (recursively). This matches the actual practice of compilers, as the ACATS has always required this behavior. 25.c/2

{*AI95-00317-01*} The description of which operations are visible in a formal package has been clarified. We also specify how matching is done when the actual is a formal package. 25.d/2

Incompatibilities With Ada 2005

{*AI05-0025-1*} {*AI05-0200-1*} **Correction:** Added missing rules for parameters of generic formal package that parallel those in 12.3, as well as some specific to <> parameters. These are technically incompatibilities because generic formal 25.e/3

package parameters that Ada 95 and Ada 2005 would have considered legal now have to be rejected. But this should not be an issue in practice as such formal parameters could not have matched any actual generics. And it is quite likely that implementations already enforce some of these rules.

Extensions to Ada 2005

25.f/3 {*AI05-0183-1*} An optional aspect_specification can be used in a formal_package_declaration. This is described in 13.1.1.

12.8 Example of a Generic Package

Examples

1 The following example provides a possible formulation of stacks by means of a generic package. The size of each stack and the type of the stack elements are provided as generic formal parameters.

2/1 *This paragraph was deleted.*

3
```
generic
   Size : Positive;
   type Item is private;
package Stack is
   procedure Push(E : in  Item);
   procedure Pop (E : out Item);
   Overflow, Underflow : exception;
end Stack;
```

4
```
package body Stack is
```

5
```
   type Table is array (Positive range <>) of Item;
   Space : Table(1 .. Size);
   Index : Natural := 0;
```

6
```
   procedure Push(E : in Item) is
   begin
      if Index >= Size then
         raise Overflow;
      end if;
      Index := Index + 1;
      Space(Index) := E;
   end Push;
```

7
```
   procedure Pop(E : out Item) is
   begin
      if Index = 0 then
         raise Underflow;
      end if;
      E := Space(Index);
      Index := Index - 1;
   end Pop;
```

8
```
end Stack;
```

9 Instances of this generic package can be obtained as follows:

10
```
package Stack_Int  is new Stack(Size => 200, Item => Integer);
package Stack_Bool is new Stack(100, Boolean);
```

11 Thereafter, the procedures of the instantiated packages can be called as follows:

12
```
Stack_Int.Push(N);
Stack_Bool.Push(True);
```

Alternatively, a generic formulation of the type Stack can be given as follows (package body omitted): 13

```
generic                                                                    14
   type Item is private;
package On_Stacks is
   type Stack(Size : Positive) is limited private;
   procedure Push(S : in out Stack; E : in  Item);
   procedure Pop (S : in out Stack; E : out Item);
   Overflow, Underflow : exception;
private
   type Table is array (Positive range <>) of Item;
   type Stack(Size : Positive) is
      record
         Space : Table(1 .. Size);
         Index : Natural := 0;
      end record;
end On_Stacks;
```

In order to use such a package, an instance has to be created and thereafter stacks of the corresponding type can be declared: 15

```
declare                                                                    16
   package Stack_Real is new On_Stacks(Real); use Stack_Real;
   S : Stack(100);
begin
   ...
   Push(S, 2.54);
   ...
end;
```

13 Representation Issues

{*8652/0009*} {*AI95-00137-01*} {*AI05-0299-1*} [This clause describes features for querying and controlling certain aspects of entities and for interfacing to hardware.] 1/3

Wording Changes from Ada 83

{*AI05-0299-1*} The subclauses of this clause have been reorganized. This was necessary to preserve a logical order, given the new Ada 95 semantics given in this section. 1.a/3

13.1 Operational and Representation Aspects

{*8652/0009*} {*AI95-00137-01*} {*AI05-0295-1*} [Two kinds of aspects of entities can be specified: representation aspects and operational aspects. Representation aspects affect how the types and other entities of the language are to be mapped onto the underlying machine. Operational aspects determine other properties of entities.] 0.1/3

Term entry: operational aspect — aspect that indicates a logical property of an entity, such as the precondition of a subprogram, or the procedure used to write a given type of object to a stream 0.a/5

Term entry: representation aspect — aspect that indicates how an entity is mapped onto the underlying hardware, for example the size or alignment of an object 0.b/5

{*AI05-0183-1*} {*AI05-0295-1*} [Either kind of aspect of an entity may be specified by means of an aspect_specification (see 13.1.1), which is an optional element of most kinds of declarations and applies to the entity or entities being declared. Aspects may also be specified by certain other constructs occurring subsequent to the declaration of the affected entity: a representation aspect value may be specified by means of a representation item and an operational aspect value may be specified by means of an operational item.] 0.2/3

{*8652/0009*} {*AI95-00137-01*} There are six kinds of *representation items*: attribute_definition_clauses for representation attributes, enumeration_representation_clauses, record_representation_clauses, at_clauses, component_clauses, and *representation pragmas*. [They can be provided to give more efficient representation or to interface with features that are outside the domain of the language (for example, peripheral hardware).] 1/1

{*8652/0009*} {*AI95-00137-01*} An *operational item* is an attribute_definition_clause for an operational attribute. 1.1/1

{*8652/0009*} {*AI95-00137-01*} [An operational item or a representation item applies to an entity identified by a local_name, which denotes an entity declared local to the current declarative region, or a library unit declared immediately preceding a representation pragma in a compilation.] 1.2/1

Language Design Principles

{*8652/0009*} {*AI95-00137-01*} {*AI05-0295-1*} Representation aspects are intended to refer to properties that need to be known before the compiler can generate code to create or access an entity. For instance, the size of an object needs to be known before the object can be created. Conversely, operational aspects are those that only need to be known before they can be used. For instance, how an object is read from a stream only needs to be known when a stream read is executed. Thus, representation aspects have stricter rules as to when they can be specified. 1.a/3

{*AI95-00291-02*} {*AI05-0295-1*} Confirming the value of an aspect should never change the semantics of the aspect. Thus Size = 8 (for example) means the same thing whether it was specified with a representation item or whether the compiler chose this value by default. 1.a.1/3

Term entry: aspect — specifiable property of an entity 1.b/5
Note: An aspect can be specified by an aspect_specification on the declaration of the entity. Some aspects can be queried via attributes.

Syntax

2/1 {*8652/0009*} {*AI95-00137-01*} aspect_clause ::= attribute_definition_clause
| enumeration_representation_clause
| record_representation_clause
| at_clause

3 local_name ::= direct_name
| direct_name'attribute_designator
| *library_unit*_name

4/1 {*8652/0009*} {*AI95-00137-01*} A representation pragma is allowed only at places where an aspect_clause or compilation_unit is allowed.

Name Resolution Rules

5/1 {*8652/0009*} {*AI95-00137-01*} In an operational item or representation item, if the local_name is a direct_name, then it shall resolve to denote a declaration (or, in the case of a pragma, one or more declarations) that occurs immediately within the same declarative region as the item. If the local_name has an attribute_designator, then it shall resolve to denote an implementation-defined component (see 13.5.1) or a class-wide type implicitly declared immediately within the same declarative region as the item. A local_name that is a *library_unit*_name (only permitted in a representation pragma) shall resolve to denote the library_item that immediately precedes (except for other pragmas) the representation pragma.

5.a/1 **Reason:** {*8652/0009*} {*AI95-00137-01*} This is a Name Resolution Rule, because we don't want an operational or representation item for X to be ambiguous just because there's another X declared in an outer declarative region. It doesn't make much difference, since most operational or representation items are for types or subtypes, and type and subtype names can't be overloaded.

5.b/1 **Ramification:** {*8652/0009*} {*AI95-00137-01*} The visibility rules imply that the declaration has to occur before the operational or representation item.

5.c/1 {*8652/0009*} {*AI95-00137-01*} For objects, this implies that operational or representation items can be applied only to stand-alone objects.

Legality Rules

6/1 {*8652/0009*} {*AI95-00137-01*} The local_name of an aspect_clause or representation pragma shall statically denote an entity (or, in the case of a pragma, one or more entities) declared immediately preceding it in a compilation, or within the same declarative_part, package_specification, task_definition, protected_definition, or record_definition as the representation or operational item. If a local_name denotes a [local] callable entity, it may do so through a [local] subprogram_renaming_declaration [(as a way to resolve ambiguity in the presence of overloading)]; otherwise, the local_name shall not denote a renaming_declaration.

6.a **Ramification:** The “statically denote” part implies that it is impossible to specify the representation of an object that is not a stand-alone object, except in the case of a representation item like pragma Atomic that is allowed inside a component_list (in which case the representation item specifies the representation of components of all objects of the type). It also prevents the problem of renamings of things like “P.**all**” (where P is an access-to-subprogram value) or “E(I)” (where E is an entry family).

6.b The part about where the denoted entity has to have been declared appears twice — once as a Name Resolution Rule, and once as a Legality Rule. Suppose P renames Q, and we have a representation item in a declarative_part whose local_name is P. The fact that the representation item has to appear in the same declarative_part as P is a Name Resolution Rule, whereas the fact that the representation item has to appear in the same declarative_part as Q is a Legality Rule. This is subtle, but it seems like the least confusing set of rules.

6.c **Discussion:** A separate Legality Rule applies for component_clauses. See 13.5.1, “Record Representation Clauses”.

7/5 {*AI95-00291-02*} {*AI12-0445-1*} The *representation* of an object consists of a certain number of bits (the *size* of the object). For an object of an elementary type, these are the bits that are normally read or

updated by the machine code when loading, storing, or operating-on the value of the object. For an object of a composite type, these are the bits reserved for this object, and include bits occupied by subcomponents of the object. If the size of an object is greater than that of its subtype, the additional bits are padding bits. For an elementary object, these padding bits are normally read and updated along with the others. For a composite object, it is unspecified whether padding bits are read or updated in any given composite operation.

To be honest: {*AI95-00291-02*} Discontiguous representations are allowed, but the ones we're interested in here are generally contiguous sequences of bits. For a discontiguous representation, the size doesn't necessarily describe the "footprint" of the object in memory (that is, the amount of space taken in the address space for the object). 7.a/2

Discussion: {*AI95-00291-02*} In the case of composite objects, we want the implementation to have the flexibility to either do operations component-by-component, or with a block operation covering all of the bits. We carefully avoid giving a preference in the wording. There is no requirement for the choice to be documented, either, as the implementation can make that choice based on many factors, and could make a different choice for different operations on the same object. 7.a.1/2

{*AI95-00291-02*} In the case of a properly aligned, contiguous object whose size is a multiple of the storage unit size, no other bits should be read or updated as part of operating on the object. We don't say this normatively because it would be difficult to normatively define "properly aligned" or "contiguous". 7.a.2/2

Ramification: Two objects with the same value do not necessarily have the same representation. For example, an implementation might represent False as zero and True as any odd value. Similarly, two objects (of the same type) with the same sequence of bits do not necessarily have the same value. For example, an implementation might use a biased representation in some cases but not others: 7.b

```
{AI05-0229-1} subtype S is Integer range 1..256;                    7.c/3
type A is array(Natural range 1..4) of S
   with Pack;
X : S := 3;
Y : A := (1, 2, 3, 4);
```

The implementation might use a biased-by-1 representation for the array elements, but not for X. X and Y(3) have the same value, but different representation: the representation of X is a sequence of (say) 32 bits: 0...011, whereas the representation of Y(3) is a sequence of 8 bits: 00000010 (assuming a two's complement representation). 7.d

Such tricks are not required, but are allowed. 7.e

Discussion: The value of any padding bits is not specified by the language, though for a numeric type, it will be much harder to properly implement the predefined operations if the padding bits are not either all zero, or a sign extension. 7.f

Ramification: {*AI05-0229-1*} For example, suppose S'Size = 2, and an object X is of subtype S. If the machine code typically uses a 32-bit load instruction to load the value of X, then X'Size should be 32, even though 30 bits of the value are just zeros or sign-extension bits. On the other hand, if the machine code typically masks out those 30 bits, then X'Size should be 2. Usually, such masking only happens for components of a composite type for which Pack, Component_Size, or record layout is specified. 7.g/3

Note, however, that the formal parameter of an instance of Unchecked_Conversion is a special case. Its Size is required to be the same as that of its subtype. 7.h

Note that we don't generally talk about the representation of a value. A value is considered to be an amorphous blob without any particular representation. An object is considered to be more concrete. 7.i

{*AI05-0112-1*} {*AI05-0295-1*} {*AI12-0396-1*} {*AI12-0407-1*} A representation item *directly specifies* a *representation aspect* of the entity denoted by the local_name, except in the case of a type-related representation item, whose local_name shall denote a first subtype, and which directly specifies an aspect of the subtype's type. A representation item that names a subtype is either *subtype-specific* (Size, Object_Size, and Alignment clauses) or *type-related* (all others). 8/5

This paragraph was deleted.{*AI12-0407-1*} 8.a/5

To be honest: Some representation items directly specify more than one aspect. 8.b

Discussion: {*AI05-0229-1*} For example, a pragma Export (see J.15.5) specifies the convention of an entity, and also specifies that it is exported. Such items are obsolescent; directly specifying the associated aspects is preferred. 8.c/3

Ramification: Each specifiable attribute constitutes a separate aspect. An enumeration_representation_clause specifies the coding aspect. A record_representation_clause (without the mod_clause) specifies the record layout aspect. Each representation pragma specifies a separate aspect. 8.d

8.e **Reason:** We don't need to say that an at_clause or a mod_clause specify separate aspects, because these are equivalent to attribute_definition_clauses. See J.7, “At Clauses”, and J.8, “Mod Clauses”.

8.e.1/3 {*AI05-0112-1*} We give a default naming for representation aspects of representation pragmas so we don't have to do that for every pragma. Operational and representation attributes are given a default naming in 13.3. We don't want any anonymous aspects; that would make other rules more difficult to write and understand.

8.f **Ramification:** The following representation items are type-related:

8.g • enumeration_representation_clause

8.h • record_representation_clause

8.i • Component_Size clause

8.j/1 • *This paragraph was deleted.*{*8652/0009*} {*AI95-00137-01*}

8.k • Small clause

8.l • Bit_Order clause

8.m • Storage_Pool clause

8.n • Storage_Size clause

8.o/2 • {*8652/0009*} {*AI95-00137-01*} {*AI95-00270-01*} Stream_Size clause

8.p/5 • {*8652/0009*} {*AI95-00137-01*} {*AI12-0005-1*} pragma Unchecked_Union

8.q/1 • *This paragraph was deleted.*{*8652/0009*} {*AI95-00137-01*}

8.r/1 • *This paragraph was deleted.*{*8652/0009*} {*AI95-00137-01*}

8.s • Machine_Radix clause

8.t • pragma Pack

8.u • pragmas Import, Export, and Convention (when applied to a type)

8.v/3 • {*AI05-0009-1*} pragmas Atomic, Independent, and Volatile (when applied to a type)

8.w/3 • {*AI05-0009-1*} pragmas Atomic_Components, Independent_Components, and Volatile_Components (when applied to a type)

8.x • pragma Discard_Names (when applied to an enumeration or tagged type)

8.x.1/5 • {*AI12-0005-1*} pragmas Priority and Interrupt_Priority (when applied to a task or protected type)

8.x.2/5 • {*AI12-0005-1*} pragma CPU (when applied to a task or protected type)

8.x.3/5 • {*AI12-0005-1*} pragma Dispatching_Domain

8.x.4/5 • {*AI12-0005-1*} pragma Relative_Deadline (when applied to a task or protected type)

8.y The following representation items are subtype-specific:

8.z • Alignment clause (when applied to a first subtype)

8.z.1/5 • {*AI12-0396-1*} Object_Size clause

8.aa • Size clause (when applied to a first subtype)

8.bb The following representation items do not apply to subtypes, so they are neither type-related nor subtype-specific:

8.cc • Address clause (applies to objects and program units)

8.dd • Alignment clause (when applied to an object)

8.ee • Size clause (when applied to an object)

8.ff/2 • {*AI95-00414-01*} pragma Inline (applies to subprograms)

8.gg • pragmas Import, Export, and Convention (when applied to anything other than a type)

8.hh • pragmas Atomic and Volatile (when applied to an object or a component)

8.ii/3 • {*AI05-0009-1*} pragmas Atomic_Components, Independent_Components, and Volatile_Components (when applied to an array object)

8.jj • pragma Discard_Names (when applied to an exception)

8.kk • pragma Asynchronous (applies to procedures)

8.ll/2 • {*AI95-00414-01*} pragma No_Return (applies to subprograms)

8.ll.1/5 • {*AI12-0005-1*} pragmas Attach_Handler and Interrupt_Handler (apply to protected units)

8.ll.2/5 • {*AI12-0005-1*} pragmas Priority and Interrupt_Priority (when applied to a subprogram)

8.ll.3/5 • {*AI12-0005-1*} pragma CPU (when applied to a subprogram)

- {*AI12-0005-1*} pragma Relative_Deadline (when applied to a subprogram) 8.ll.4/5

{*AI05-0229-1*} {*AI12-0005-1*} While an aspect_specification is not a representation item, a similar categorization applies to the aspect that corresponds to each of these representation items. In addition, there are representation aspects that do not have associated representation items. 8.mm/5

{*AI12-0005-1*} The following representation aspects that do not have associated representation items are type-related: 8.mm.1/5

- aspect Default_Value 8.mm.2/5
- aspect Default_Component_Value 8.mm.3/5
- aspect Exclusive_Functions 8.mm.4/5
- aspects Link_Name and External_Name (when applied to a type) 8.mm.5/5
- aspect Max_Entry_Queue_Length (when applied to a task or protected type) 8.mm.6/5

{*AI12-0005-1*} There are no subtype-specific representation aspects that do not have associated representation items. 8.mm.7/5

{*AI12-0005-1*} The following representation aspects that do not have associated representation items also do not apply to subtypes, so they are neither type-related nor subtype-specific: 8.mm.8/5

- aspect Static (applies to expression functions) 8.mm.9/5
- aspect Synchronization (applies to procedures) 8.mm.10/5
- aspects Link_Name and External_Name (when applied to anything other than a type) 8.mm.11/5
- aspect Max_Entry_Queue_Length (when applied to an entry) 8.mm.12/5

{*8652/0009*} {*AI95-00137-01*} {*AI05-0183-1*} An operational item *directly specifies* an *operational aspect* of the entity denoted by the local_name, except in the case of a type-related operational item, whose local_name shall denote a first subtype, and which directly specifies an aspect of the type of the subtype. 8.1/3

Ramification: {*8652/0009*} {*AI95-00137-01*} The following operational items are type-related: 8.nn/1

- External_Tag clause 8.oo/1
- Read clause 8.pp/1
- Write clause 8.qq/1
- Input clause 8.rr/1
- Output clause 8.ss/1

{*AI12-0407-1*} Aspects that can be specified for types and subtypes are also classified into type-related or subtype-specific aspects. Representation aspects that can be specified for types and subtypes are considered type-related unless specified otherwise. In contrast, the classification of operational aspects are given with the definition of the aspect. Type-related aspects have the same value for all subtypes of (a view of) a type[, while subtype-specific aspects may differ for different subtypes of the same type]. 8.2/5

Discussion: We talk about “type-related aspects”, which of course include both type-related representation aspects and type-related operational aspects. Aspects Size, Object_Size, and Alignment are subtype-specific, as that is specified above. All other representation aspects are either type-related or not specifiable for a type or subtype. Unlike representation aspects, there is no default for operational aspects; whether they are type-related or subtype-specific should be part of the definition of the aspect. 8.tt/5

{*AI05-0183-1*} {*AI12-0116-1*} {*AI12-0181-1*} {*AI12-0222-1*} A representation item or operational item that directly specifies an aspect of an entity shall appear before the entity is frozen (see 13.14). 9/5

Ramification: {*8652/0009*} {*AI95-00137-01*} {*AI12-0222-1*} The fact that a representation item (or operational item) that directly specifies an aspect of an entity is required to appear before the entity is frozen prevents changing the representation of an entity after using the entity in ways that require the representation to be known. 9.a/5

This paragraph was deleted.{*AI05-0183-1*} {*AI12-0116-1*} 9.b/4

{*8652/0009*} {*AI95-00137-01*} {*AI05-0183-1*} {*AI12-0116-1*} {*AI12-0222-1*} A representation aspect of a subtype or type shall not be specified (whether by a representation item or an aspect_specification) before the type is completely defined (see 3.11.1). 9.1/5

Ramification: Unlike representation items, operational items can be specified on partial views. Since they don't affect the representation, the full declaration need not be known to determine their legality. 9.c/1

9.2/5 {*AI12-0116-1*} {*AI12-0396-1*} {*AI12-0417-1*} If a representation item, operational item, library unit pragma (see J.15), or aspect_specification is given that directly specifies an aspect of an entity, then it is illegal to give another representation item, operational item, library unit pragma, or aspect_specification that directly specifies the same aspect of the entity.

9.c.1/5 **Ramification:** {*AI12-0116-1*} {*AI12-0396-1*} This rule applies to all aspects, not just those that have operational items or representation items. For instance, it applies to subtype predicates and type invariants.

9.c.2/4 **To be honest:** {*AI12-0116-1*} This rule is also intended to cover other ways to specify representation aspects, such as obsolescent pragma Priority. Priority is not a representation pragma, and as such is neither a representation item nor an aspect_specification. Regardless, giving both a pragma Priority and an aspect_specification for Priority is illegal. We didn't want to complicate the wording solely to support obsolescent features.

9.3/5 {*AI05-0106-1*} {*AI05-0295-1*} {*AI12-0116-1*} {*AI12-0181-1*} Unless otherwise specified, it is illegal to specify an operational or representation aspect of a generic formal parameter.

9.d/5 **Reason:** {*AI12-0064-2*} Specifying an aspect on a generic formal parameter implies an added contract for a generic unit. That contract needs to be defined via generic parameter matching rules, and, as aspects vary widely, that has to be done for each such aspect. Since most aspects do not need this complexity, we avoid the complexity by saying that such contract-forming aspect specifications are banned unless the rules defining them explicitly exist. Note that the method of specification does not matter: aspect_specifications, representation items, and operational items are all covered by this rule.

10/5 {*AI05-0295-1*} {*AI12-0109-1*} {*AI12-0376-1*} {*AI12-0427-1*} A *by-reference primitive* is a user-defined primitive subprogram for a type *T* that has an access result designating type *T*, or that has a formal parameter that is an access parameter designating type *T* or is aliased and of type *T*. It is illegal to specify a nonconfirming type-related representation aspect for an untagged type *T* if it is derived from a by-reference type or inherits one or more by-reference primitives, or if one or more types have been derived from *T* prior to the specification of the aspect and type *T* is a by-reference type or defines one or more by-reference primitives that are inherited by these descendants.

10.a/3 **Ramification:** {*8652/0009*} {*AI95-00137-01*} {*AI05-0295-1*} On the other hand, subtype-specific representation aspects may be specified for the first subtype of such a type, as can operational aspects.

10.b/5 **Reason:** {*AI05-0229-1*} {*AI05-0295-1*} {*AI12-0109-1*} {*AI12-0376-1*} {*AI12-0427-1*} The reason for forbidding specification of type-related representation aspects on untagged by-reference types is because a change of representation is impossible when passing by reference (to an inherited subprogram). (A by-reference object cannot be copied to change its representation.) The reason for forbidding specification of type-related representation aspects on untagged types with by-reference primitives is that access parameters, access results, and aliased parameters cannot be converted as part of invoking an inherited subprogram if the representation of the designated types might be different. This rule is not needed for tagged types, because other rules prevent a type-related representation aspect from changing the representation of the parent part; we want to allow specifying a type-related representation aspect on a type extension to specify aspects of the extension part. For example, specifying aspect Pack will cause packing of the extension part, but not of the parent part.

10.c/4 **Discussion:** {*AI12-0109-1*} "By-reference type" usually cannot be used in Legality Rules, as it is privacy breaking. Our use here is privacy breaking, but we're stuck with it for compatibility reasons. Since representation aspects cannot be specified on partial views, privacy violations only can happen when a type includes a component of a private type. In that case, whether these rules are triggered depends on the full type of the private type — which is clearly privacy breaking.

11/5 {*8652/0009*} {*AI95-00137-01*} {*8652/0011*} {*AI95-00117-01*} {*AI95-00326-01*} {*AI05-0295-1*} {*AI12-0396-1*} {*AI12-0409-1*} If a type-related aspect is defined for the partial view of a type, then it has the same definition for the full view of the type, except for certain Boolean-valued operational aspects where the language specifies that the partial view can have the value False even when the full view has the value True. Type-related aspects [cannot be specified, and] are not defined for an incomplete view of a type. Representation aspects of a generic formal parameter are the same as those of the actual. Specification of a type-related representation aspect is not allowed for a descendant of a generic formal untagged type.

11.a/3 **Ramification:** {*8652/0009*} {*AI95-00137-01*} {*AI05-0295-1*} Specifying representation aspects is allowed for types whose subcomponent types or index subtypes are generic formal types. Specifying operational aspects and subtype-related representation aspects is allowed on descendants of generic formal types.

Reason: {*AI05-0295-1*} Since it is not known whether a formal type has user-defined primitive subprograms, specifying type-related representation aspects for them is not allowed, unless they are tagged (in which case only the extension part is affected in any case). 11.b/3

This paragraph was deleted.{*AI95-00326-01*} {*AI12-0396-1*} 11.c/5

Ramification: {*AI05-0083-1*} {*AI12-0396-1*} Different views of an entity can have different representation and operational aspects. Most type-related aspects are the same for all views of a type; but specific aspects may have different rules. Similarly, different views of an object can have different representation aspects. For instance, an actual object passed by reference and the associated formal parameter may have different values for Alignment even though the formal parameter is merely a view of the actual object. This is necessary to maintain the language design principle that Alignments are always known at compile time. 11.d/5

Discussion: {*AI12-0409-1*} For type-related aspects, the aspect of the partial view (if any) and full view have to be the same, with an exception for certain Boolean-valued aspects. The exception is intended for aspect Preelaborable_Initialization, where it would be wildly incompatible (as well as unfriendly) to require the (possibly implicit) state of the full view to be repeated explicitly on the partial view. Aspect Nonblocking has a similar rule for similar reasons, though it is a subtype-specific aspect. 11.e/5

Proof: {*AI12-0396-1*} The syntax of incomplete_type_declarations and formal_incomplete_type_declarations do not allow for an aspect_specification. The prefix of an attribute_reference cannot name an incomplete type (see 3.10.1), so no attribute_definition_clause can be used with an incomplete type. Similarly, the local_name of a representation pragma cannot name an incomplete type, so no such pragmas can be used with an incomplete type. 11.f/5

{*AI05-0295-1*} The specification of the Size aspect for a given subtype, or the size or storage place for an object (including a component) of a given subtype, shall allow for enough storage space to accommodate any value of the subtype. 12/3

{*8652/0009*} {*AI95-00137-01*} {*AI05-0295-1*} {*AI12-0396-1*} {*AI12-0444-1*} The specification of certain language-defined aspects is not required to be supported by all implementations; in such an implementation, the specification for such an aspect is illegal or raises an exception at run time. 13/5

Ramification: There is an Implementation Permission below that allows an implementation to put restrictions on any representation aspect; this rule applies if that permission is used for an aspect. This rule only applies to an operational aspect if the aspect explicitly allows the aspect to not be supported. 13.a

{*AI95-00251-01*} {*AI05-0295-1*} A type_declaration is illegal if it has one or more progenitors, and a nonconfirming value was specified for a representation aspect of an ancestor, and this conflicts with the representation of some other ancestor. The cases that cause conflicts are implementation defined. 13.1/3

Implementation defined: The cases that cause conflicts between the representation of the ancestors of a type_declaration. 13.b/2

Reason: {*AI05-0295-1*} This rule is needed because it may be the case that only the combination of types in a type declaration causes a conflict. Thus it is not possible, in general, to reject the original representation item or aspect_specification. For instance: 13.c/3

```
package Pkg1 is                                                    13.d/2
   type Ifc is interface;
   type T is tagged record
      Fld : Integer;
   end record;
   for T use record
      Fld at 0 range 0 .. Integer'Size - 1;
   end record;
end Pkg1;
```

Assume the implementation uses a single tag with a default offset of zero, and that it allows the use of nondefault locations for the tag (and thus accepts representation items like the one above). The representation item will force a nondefault location for the tag (by putting a component other than the tag into the default location). Clearly, this package will be accepted by the implementation. However, other declarations could cause trouble. For instance, the implementation could reject: 13.e/2

```
with Pkg1;                                                         13.f/2
package Pkg2 is
   type NewT is new Pkg1.T and Pkg1.Ifc with null record;
end Pkg2;
```

{*AI05-0295-1*} because the declarations of T and Ifc have a conflict in their representation items. This is clearly necessary (it's hard to imagine how Ifc'Class could work with the tag at a location other than the one it is expecting without introducing distributed overhead). 13.g/3

13.h/3 {*AI05-0295-1*} Conflicts will usually involve implementation-defined attributes (for specifying the location of the tag, for instance), although the example above shows that doesn't have to be the case. For this reason, we didn't try to specify exactly what causes a conflict; it will depend on the implementation's implementation model and what representation aspects it allows to be changed.

13.i/3 **Implementation Note:** {*AI05-0295-1*} An implementation can only use this rule to reject type_declarations where one of its ancestors had a nonconfirming representation value specified. An implementation must ensure that the default representations of ancestors cannot conflict.

13.2/5 {*AI12-0427-1*} When specifying an aspect that denotes a subprogram, the profile of the subprogram shall be mode conformant with the one required for the aspect, and the convention shall be Ada. Additional requirements are defined for particular aspects.

13.j/5 **Ramification:** As well as applying to aspect_specifications, this rule applies to attribute_definition_clauses for those aspects that have associated attributes.

13.k/5 This rule implies, for example, that if one writes:

13.l/5
```
type T is ...
   with Read => R;
```

13.m/5 R has to be a procedure with two parameters with the appropriate subtypes and modes as shown in 13.13.2.

Static Semantics

14/5 {*AI12-0059-1*} If two subtypes statically match, then their subtype-specific aspects (for example, Size and Alignment) are the same.

14.a/5 **Reason:** {*AI05-0295-1*} {*AI12-0059-1*} This is necessary because we allow (for example) conversion between access types whose designated subtypes statically match. Note that most aspects (including the subtype-specific aspects Size and Alignment) may not be specified for a nonfirst subtype. The only language-defined exceptions to this rule are the Object_Size, Static_Predicate, and Dynamic_Predicate aspects.

14.b Consider, for example:

14.c/5
```
{AI12-0005-1} package P1 is
   subtype S1 is Integer range 0..2**16-1
      with Size => 16; -- Illegal!
      -- S1'Size would be 16 by default.
   type A1 is access all S1;
   X1: A1;
end P1;
```

14.d/5
```
{AI12-0005-1} package P2 is
   subtype S2 is Integer range 0..2**16-1
      with Size => 32; -- Illegal!
   type A2 is access all S2;
   X2: A2;
end P2;
```

14.e/3
```
{AI05-0229-1} procedure Q is
   use P1, P2;
   type Array1 is array(Integer range <>) of aliased S1
      with Pack;
   Obj1: Array1(1..100);
   type Array2 is array(Integer range <>) of aliased S2
      with Pack;
   Obj2: Array2(1..100);
begin
   X1 := Obj2(17)'Unchecked_Access;
   X2 := Obj1(17)'Unchecked_Access;
end Q;
```

14.f Loads and stores through X1 would read and write 16 bits, but X1 points to a 32-bit location. Depending on the endianness of the machine, loads might load the wrong 16 bits. Stores would fail to zero the other half in any case.

14.g Loads and stores through X2 would read and write 32 bits, but X2 points to a 16-bit location. Thus, adjacent memory locations would be trashed.

14.h Hence, the above is illegal. Furthermore, the compiler is forbidden from choosing different Sizes by default, for the same reason.

14.i/5 {*AI12-0059-1*} The same issues apply to Alignment. Similar issues apply to Object_Size, Static_Predicate, and Dynamic_Predicate, but the chosen solution is different: explicit rules in 4.9.1 that ensure that the aspects are the same if specified.

{*AI12-0059-1*} The above static matching rule combined with the definition of static matching in 4.9.1 (after updating to add Object_Size) does not cause incompatibilities in existing Ada 2012 code that does not mention Object_Size. But it does constrain the implementation-defined value of Object_Size when it is not specified for a subtype; the above rule applies even in that case. (The effects noted in the example above would occur otherwise.) 14.j/5

{*AI12-0059-1*} We need this rule even though static matching explicitly excludes confirming values of Object_Size. That's because a general access type can designate any aliased object whose subtype statically matches the designated subtype. Since the Object_Size of a subtype determines the number of bits allocated for an aliased object of the subtype, if we allowed different Object_Sizes for statically matching subtypes, we'd be allowing the access type to designate objects with differing numbers of bits. That isn't going to work. 14.k/5

{*8652/0040*} {*AI95-00108-01*} {*AI05-0009-1*} {*AI05-0295-1*} A derived type inherits each type-related representation aspect of its parent type that was directly specified before the declaration of the derived type, or (in the case where the parent is derived) that was inherited by the parent type from the grandparent type. A derived subtype inherits each subtype-specific representation aspect of its parent subtype that was directly specified before the declaration of the derived type, or (in the case where the parent is derived) that was inherited by the parent subtype from the grandparent subtype, but only if the parent subtype statically matches the first subtype of the parent type. An inherited representation aspect is overridden by a subsequent aspect_specification or representation item that specifies a different value for the same aspect of the type or subtype. 15/3

To be honest: A record_representation_clause for a record extension does not override the layout of the parent part; if the layout was specified for the parent type, it is inherited by the record extension. 15.a

Ramification: If a representation item for the parent appears after the derived_type_definition, then inheritance does not happen for that representation item. 15.b

{*AI05-0009-1*} {*AI05-0295-1*} If an inherited aspect is confirmed by an aspect_specification or a later representation item for a derived type, the confirming specification does not override the inherited one. Thus the derived type has both a specified confirming value and an inherited nonconfirming representation value — this means that rules that apply only to nonconfirming representation values still apply to this type. 15.b.1/3

{*AI12-0109-1*} If an aspect was specified by an aspect_specification and the parent type has not yet been frozen, then the inherited aspect might not yet have been resolved and evaluated. The implementation will need to have a mechanism to handle such an aspect. 15.b.2/4

{*8652/0040*} {*AI95-00108-01*} {*AI95-00444-01*} {*AI05-0183-1*} {*AI05-0295-1*} {*AI12-0396-1*} In contrast, whether type-related operational aspects are inherited by a derived type depends on each specific aspect; unless specified, an operational aspect is not inherited. When type-related operational aspects are inherited by a derived type, aspects that were directly specified by aspect_specifications or operational items that are visible at any point within the immediate scope of the derived type declaration, or (in the case where the parent is derived) that were inherited by the parent type from the grandparent type, are inherited. An inherited operational aspect is overridden by an aspect_specification or operational item that specifies the same aspect of the type. 15.1/5

Ramification: As with representation items, if an operational item for the parent appears after the derived_type_-definition, then inheritance does not happen for that operational item. 15.b.3/1

{*AI12-0396-1*} Unlike representation aspects, operational aspects can be inherited at any point in the immediate scope, and can be overridden by aspect_specifications or operational items that occur at a point before or after the inheritance takes place. 15.b.4/5

{*AI12-0396-1*} When a type-related operational aspect is inherited, the rules for inheritance depend on the nature of the aspect (see 13.1.1). Unless otherwise specified for a given aspect, these rules are as follows: 15.2/5

- For an operational aspect that is a value, the inherited aspect has the same value; 15.3/5
- For an operational aspect that is a name: 15.4/5
  - if the name denotes one or more primitive subprograms of the type, the inherited aspect is a name that denotes the corresponding primitive subprogram(s) of the derived type; 15.5/5

Discussion: {*AI12-0005-1*} A primitive subprogram that is declared after the end of the enclosing declaration list of a derived type (in particular, in the private part of a package if the derived type is declared in the package's visible part) 15.b.6/5

is not the “corresponding primitive subprogram” in the sense of this rule. In a case like this, the corresponding primitive subprogram is actually the inherited subprogram. This doesn't make a difference for tagged types, because of the “dispatching” semantics used, which means a call on the inherited subprogram actually “reaches” the body of the overriding. But for untagged types, overriding an inherited subprogram of an untagged derived type in the private part is “too late” to have an effect on external uses of the primitive, since non-visible overridings are not generally reachable, and a call that only “sees” the inherited subprogram goes to the body of the parent's operation, effectively ignoring the private overriding. This same rule applies to aspects that denote an inherited subprogram of an untagged derived type — the private overriding is effectively ignored.

15.6/5 • otherwise, the inherited aspect is a name that denotes the same entity or entities as the original aspect;

15.7/5 • {*AI12-0396-1*} {*AI12-0423-1*} For an operational aspect that is an identifier specific to the aspect, the inherited aspect is the same identifier;

15.8/5 • {*AI12-0396-1*} {*AI12-0423-1*} For an operational aspect that is an expression or an aggregate, the inherited aspect is a corresponding expression or aggregate where each name, value, and identifier follows these same rules for inheritance.

16 Each aspect of representation of an entity is as follows:

17 • If the aspect is *specified* for the entity, meaning that it is either directly specified or inherited, then that aspect of the entity is as specified, except in the case of Storage_Size, which specifies a minimum.

17.a **Ramification:** This rule implies that queries of the aspect return the specified value. For example, if the user writes “**for** X'Size **use** 32;”, then a query of X'Size will return 32.

18 • If an aspect of representation of an entity is not specified, it is chosen by default in an unspecified manner.

18.a/3 **Ramification:** {*8652/0009*} {*AI95-00137-01*} {*AI05-0295-1*} Note that specifying a representation aspect can affect the semantics of the entity.

18.b The rules forbid things like “**for** S'Base'Alignment **use** ...” and “**for** S'Base **use** record ...”.

18.c **Discussion:** The intent is that implementations will represent the components of a composite value in the same way for all subtypes of a given composite type. Hence, Component_Size and record layout are type-related aspects.

18.d/3 **Ramification:** {*AI05-0083-1*} As noted previously, in the case of an object, the entity mentioned in this text is a specific view of an object. That means that only references to the same view of an object that has a specified value for a representation aspect *R* necessarily have that value for the aspect *R*. The value of the aspect *R* for a different view of that object is unspecified. In particular, this means that the representation values for by-reference parameters is unspecified; they do not have to be the same as those of the underlying object.

18.1/5 {*8652/0040*} {*AI95-00108-01*} {*AI12-0396-1*} If an operational aspect is *specified* for an entity (meaning that it is either directly specified or, if type-related or subtype-specific, inherited), then that aspect of the entity is as specified. Otherwise, the aspect of the entity has the default value for that aspect.[For aspects that are neither type-related nor subtype-specific, the terms “specified” and “directly specified” are equivalent.]

18.e/5 **Proof:** {*AI12-0396-1*} Aspect inheritance is only defined for types and subtypes.

18.2/5 {*AI95-00291-02*} {*AI05-0295-1*} {*AI12-0396-1*} An aspect_specification or representation item that specifies a representation aspect that would have been chosen in the absence of the aspect_specification or representation item is said to be *confirming*. The aspect value specified in this case is said to be a *confirming* representation aspect value. Other values of the aspect are said to be *nonconfirming*, as are the aspect_specifications and representation items that specified them. Similarly, an aspect_specification or operational item that specifies an operational aspect to be the same as the definition it would have by default is said to be *confirming*; otherwise it is *nonconfirming*.

18.f/5 **Discussion:** {*AI12-0005-1*} As noted at the beginning of this subclause, we have a Language Design Principle that confirming aspects do not change the semantics. However, it is not a Language Design Principle principle that one can always specify a confirming aspect. Some aspects can never be confirmed. For instance, the default implementation of a stream-oriented attribute cannot be confirmed, as any subprogram that can be specified is not the same as the default

subprogram, and thus is nonconfirming. Whether the effect is the same is not relevant for this purpose; it would be very difficult to formally define what it means to have the same effect. (Note that one can name any implementation for a stream-oriented attribute by using the associated attribute but specifying that by name would violate the circular aspect definition rule.) This is true for any implicitly composed aspect. Similarly, aspect Default_Value cannot be confirmed as the default is that there is no default value; any value that can be specified is not that.

Dynamic Semantics

{*8652/0009*} {*AI95-00137-01*} For the elaboration of an aspect_clause, any evaluable constructs within it are evaluated. 19/1

Ramification: {*AI05-0299-1*} Elaboration of representation pragmas is covered by the general rules for pragmas in 2.8. 19.a/3

Implementation Permissions

{*AI05-0295-1*} An implementation may interpret representation aspects in an implementation-defined manner. An implementation may place implementation-defined restrictions on the specification of representation aspects. A *recommended level of support* is defined for the specification of representation aspects and related features in each subclause. These recommendations are changed to requirements for implementations that support the Systems Programming Annex (see C.2, "Required Representation Support"). 20/3

Implementation defined: The interpretation of each representation aspect. 20.a/3

Implementation defined: Any restrictions placed upon the specification of representation aspects. 20.b/3

Ramification: Implementation-defined restrictions may be enforced either at compile time or at run time. There is no requirement that an implementation justify any such restrictions. They can be based on avoiding implementation complexity, or on avoiding excessive inefficiency, for example. 20.c

{*8652/0009*} {*AI95-00137-01*} There is no such permission for operational aspects. 20.c.1/1

Implementation Advice

{*AI05-0295-1*} The recommended level of support for the specification of all representation aspects is qualified as follows: 21/3

- {*AI95-00291-02*} {*AI05-0295-1*} A confirming specification for a representation aspect should be supported. 21.1/3

  To be honest: {*AI05-0295-1*} A confirming representation aspect value might not be possible for some entities. For instance, consider an unconstrained array. The size of such a type is implementation-defined, and might not actually be a representable value, or might not be static. 21.a/3

- {*AI05-0295-1*} {*AI12-0444-1*} An implementation is not required to support the specification for a representation aspect that contains nonstatic expressions, unless each nonstatic expression is a name that statically denotes a constant declared before the entity. 22/5

  Reason: This is to avoid the following sort of thing: 22.a

  ```
  {AI12-0005-1} X : Integer := F(...)                                    22.b/5
     with Address => Y;
  Y : Address := G(...);
  ```

  In the above, we have to evaluate the initialization expression for X before we know where to put the result. This seems like an unreasonable implementation burden. 22.c

  The above code should instead be written like this: 22.d

  ```
  {AI12-0005-1} Y : constant Address := G(...);                          22.e/5
  X : Integer := F(...)
     with Address => Y;
  ```

  This allows the expression "Y" to be safely evaluated before X is created. 22.f

  The constant could be a formal parameter of mode **in**. 22.g

  An implementation can support other nonstatic expressions if it wants to. Expressions of type Address are hardly ever static, but their value might be known at compile time anyway in many cases. 22.h

23/5 • {*AI12-0059-1*} {*AI12-0444-1*} An implementation is not required to support a specification for the Object_Size or Size for a given composite subtype, nor the size or storage place for an object (including a component) of a given composite subtype, unless the constraints on the subtype and its composite subcomponents (if any) are all static constraints.

23.a/5 **Reason:** {*AI12-0059-1*} We don't want to require that Object_Size or Size be supported on types that might have a representation not completely specified at compile time, or are represented discontiguously, or are represented differently for different constraints.

24/5 • {*AI95-00291-02*} {*AI05-0295-1*} {*AI12-0444-1*} An implementation is not required to support specifying a nonconfirming representation aspect value if it can cause an aliased object or an object of a by-reference type to be allocated at a nonaddressable location or, when the alignment attribute of the subtype of such an object is nonzero, at an address that is not an integral multiple of that alignment.

24.a/1 **Reason:** The intent is that access types, type System.Address, and the pointer used for a by-reference parameter should be implementable as a single machine address — bit-field pointers should not be required. (There is no requirement that this implementation be used — we just want to make sure it's feasible.)

24.b/2 **Implementation Note:** {*AI95-00291-02*} We want subprograms to be able to assume the properties of the types of their parameters inside of subprograms. While many objects can be copied to allow this (and thus do not need limitations), aliased or by-reference objects cannot be copied (their memory location is part of their identity). Thus, the above rule does not apply to types that merely allow by-reference parameter passing; for such types, a copy typically needs to be made at the call site when a bit-aligned component is passed as a parameter.

25/5 • {*AI95-00291-02*} {*AI05-0295-1*} {*AI12-0444-1*} An implementation is not required to support specifying a nonconfirming representation aspect value if it can cause an aliased object of an elementary type to have a size other than that which would have been chosen by default.

25.a/2 **Reason:** Since all bits of elementary objects participate in operations, aliased objects must not have a different size than that assumed by users of the access type.

26/5 • {*AI95-00291-02*} {*AI05-0295-1*} {*AI12-0444-1*} An implementation is not required to support specifying a nonconfirming representation aspect value if it can cause an aliased object of a composite type, or an object whose type is by-reference, to have a size smaller than that which would have been chosen by default.

26.a/2 **Reason:** Unlike elementary objects, there is no requirement that all bits of a composite object participate in operations. Thus, as long as the object is the same or larger in size than that expected by the access type, all is well.

26.b/2 **Ramification:** This rule presumes that the implementation allocates an object of a size specified to be larger than the default size in such a way that access of the default size suffices to correctly read and write the value of the object.

27/5 • {*AI95-00291-02*} {*AI05-0295-1*} {*AI12-0444-1*} An implementation is not required to support specifying a nonconfirming subtype-specific representation aspect value for an indefinite or abstract subtype.

27.a/3 **Reason:** {*AI05-0295-1*} Representation aspects are often not well-defined for such types.

27.b/3 **Ramification:** {*AI95-00291-02*} {*AI05-0229-1*} A type with the Pack aspect specified will typically not be packed so tightly as to disobey the above rules. A Component_Size clause or record_representation_clause will typically be illegal if it disobeys the above rules. Atomic components have similar restrictions (see C.6, “Shared Variable Control”).

28/5 {*AI95-00291-02*} {*AI05-0295-1*} {*AI12-0444-1*} For purposes of these rules, the determination of whether specifying a representation aspect value for a type *can cause* an object to have some property is based solely on the properties of the type itself, not on any available information about how the type is used. In particular, it presumes that minimally aligned objects of this type can be declared at some point.

28.a/2 **Implementation Advice:** The recommended level of support for all representation items should be followed.

29/3 NOTE {*AI05-0229-1*} Aspects that can be specified are defined throughout this document, and are summarized in K.1.

Incompatibilities With Ada 83

29.a It is now illegal for a representation item to cause a derived by-reference type to have a different record layout from its parent. This is necessary for by-reference parameter passing to be feasible. This only affects programs that specify the

representation of types derived from types containing tasks; most by-reference types are new to Ada 95. For example, if A1 is an array of tasks, and A2 is derived from A1, it is illegal to apply a pragma Pack to A2.

Extensions to Ada 83

{*8652/0009*} {*AI95-00137-01*} Ada 95 allows additional aspect_clauses for objects. 29.b/1

Wording Changes from Ada 83

{*8652/0009*} {*AI95-00137-01*} The syntax rule for type_representation_clause is removed; the right-hand side of that rule is moved up to where it was used, in aspect_clause. There are two references to "type representation clause" in RM83, both in Section 13; these have been reworded. Also, the representation_clause has been renamed the aspect_clause to reflect that it can be used to control more than just representation aspects. 29.c/1

{*8652/0009*} {*AI95-00137-01*} {*AI95-00114-01*} We have defined a new term "representation item", which includes all representation clauses and representation pragmas, as well as component_clauses. This is convenient because the rules are almost identical for all of them. We have also defined the new terms "operational item" and "operational aspects" in order to conveniently handle new types of specifiable entities. 29.d/2

All of the forcing occurrence stuff has been moved into its own subclause (see 13.14), and rewritten to use the term "freezing". 29.e

RM83-13.1(10) requires implementation-defined restrictions on representation items to be enforced at compile time. However, that is impossible in some cases. If the user specifies a junk (nonstatic) address in an address clause, and the implementation chooses to detect the error (for example, using hardware memory management with protected pages), then it's clearly going to be a run-time error. It seems silly to call that "semantics" rather than "a restriction". 29.f

RM83-13.1(10) tries to pretend that representation_clauses don't affect the semantics of the program. One counter-example is the Small clause. Ada 95 has more counter-examples. We have noted the opposite above. 29.g

Some of the more stringent requirements are moved to C.2, "Required Representation Support". 29.h

Extensions to Ada 95

{*AI95-00291-02*} **Amendment Correction:** Confirming representation items are defined, and the recommended level of support is now that they always be supported. 29.i/2

Wording Changes from Ada 95

{*8652/0009*} {*AI95-00137-01*} **Corrigendum:** Added operational items in order to eliminate unnecessary restrictions and permissions on stream attributes. As part of this, representation_clause was renamed to aspect_clause. 29.j/2

{*8652/0009*} {*AI95-00137-01*} {*AI95-00326-01*} **Corrigendum:** Added wording to say that the partial and full views have the same operational and representation aspects. Ada 2005 extends this to cover all views, including the incomplete view. 29.k/2

{*8652/0040*} {*AI95-00108-01*} **Corrigendum:** Changed operational items to have inheritance specified for each such aspect. 29.l/2

{*AI95-00251-01*} Added wording to allow the rejection of types with progenitors that have conflicting representation items. 29.m/2

{*AI95-00291-02*} The description of the representation of an object was clarified (with great difficulty reaching agreement). Added wording to say that representation items on aliased and by-reference objects never need be supported if they would not be implementable without distributed overhead even if other recommended level of support says otherwise. This wording matches the rules with reality. 29.n/2

{*AI95-00444-01*} {*AI05-0005-1*} Added wording so that inheritance depends on whether operational items are visible rather than whether they occur before the declaration (we don't want to look into private parts). Also limited operational inheritance to untagged types to avoid anomalies with private extensions (this is not incompatible, no existing operational attribute used this capability). Also added wording to clearly define that subprogram inheritance works like derivation of subprograms. 29.o/3

Incompatibilities With Ada 2005

{*AI05-0106-1*} {*AI12-0064-2*} {*AI12-0396-1*} **Correction:** Specifying a language-defined aspect for a generic formal parameter is no longer allowed. Most aspects could not be specified on these anyway; moreover, this was not allowed in Ada 83, so it is unlikely that compilers are supporting this as a capability (and it is not likely that they have a consistent definition of what it means if it is allowed). Thus, we expect this to occur rarely in existing programs. Note: This correction is repealed in Ada 2022, as a number of aspects are allowed on formals in that later language version. 29.p/5

Wording Changes from Ada 2005

29.q/5 {*AI05-0009-1*} {*AI12-0005-1*} **Correction:** Defined that overriding of a representation aspect only happens for a nonconfirming representation item. This prevents a derived type from being considered to have only a confirming representation item when the value would be nonconfirming if given on a type that does not inherit any aspects of representation. This change just eliminates a wording confusion and ought not change any behavior.

29.r/3 {*AI05-0112-1*} **Correction:** Defined a default naming for representation aspects that are representation pragmas.

29.s/3 {*AI05-0183-1*} Added text ensuring that the rules for representational and operational items also apply appropriately to aspect_specifications; generalized operational aspects so that they can be defined for entities other than types. Any extensions are documented elsewhere.

29.t/3 {*AI05-0295-1*} Rewrote many rules to be in terms of "specifying a representation aspect" rather than use of a "representation item". This better separates *how* an aspect is specified from *what* rules apply to the value of the aspect.

Incompatibilities With Ada 2012

29.u/4 {*AI12-0109-1*} **Corrigendum:** Added a rule that makes it illegal to specify a representation value after a type is derived from an untagged by-reference type. This restriction is incompatible, but since the implementation would have had to copy an object that does not allow copying in order to change the representation for any implicit or explicit conversion between the original and the derived type, it is unlikely that any program could exist without running into internal compiler errors or bogus results.

29.v/5 {*AI12-0181-1*} **Correction:** Added a rule preventing self-referencing representation or operational aspects. This is technically incompatible, but since the similar representation or operational item has been illegal since they've existed, and they don't make any sense, it's unlikely that any implementation accepted them or that they appear in any program.

29.w/5 {*AI12-0222-1*} **Correction:** Added a rule preventing giving a representation aspect on a partial view. Since this has always been prohibited for other kinds of representation items, it's unlikely that any implementation accepted them or that they appear in any program.

Extensions to Ada 2012

29.x/5 {*AI12-0376-1*} It is now allowed to specify type-related representation aspects for an untagged derived type that has primitive operations, so long as the type is not a by-reference type.

Wording Changes from Ada 2012

29.y/5 {*AI12-0116-1*} {*AI12-0396-1*} **Corrigendum:** Clarified that an aspect (any aspect) can be specified only once for an entity, no matter what means of specifying it are used.

29.z/5 {*AI12-0059-1*} Added wording changes to support Object_Size.

29.aa/5 {*AI12-0396-1*} **Correction:** Clarified that many of these rules only make sense for type-related aspects.

29.bb/5 {*AI12-0396-1*} {*AI12-0419-1*} {*AI12-0423-1*} **Correction:** Clarified how inheritance works for operational aspects that are not values.

29.cc/5 {*AI12-0427-1*} Move the general requirements for aspects that are subprograms here from 13.3.

13.1.1 Aspect Specifications

1/3 {*AI05-0183-1*} [Certain representation or operational aspects of an entity may be specified as part of its declaration using an aspect_specification, rather than using a separate representation or operational item.] The declaration with the aspect_specification is termed the *associated declaration*.

Syntax

2/3 {*AI05-0183-1*} aspect_specification ::=
with aspect_mark [=> aspect_definition] {,
aspect_mark [=> aspect_definition] }

3/3 {*AI05-0183-1*} aspect_mark ::= *aspect*_identifier['Class]

4/5 {*AI05-0183-1*} {*AI12-0187-1*} {*AI12-0285-1*} {*AI12-0373-1*} aspect_definition ::=
name | expression | identifier
| aggregate | global_aspect_definition

Language Design Principles

{*AI05-0183-1*} {*AI05-0267-1*} The aspect_specification is an optional element in most kinds of declarations. Here is a list of all kinds of declarations and an indication of whether or not they allow aspect clauses, and in some cases a short discussion of why (* = allowed, NO = not allowed). Kinds of declarations with no indication are followed by their subdivisions (which have indications). 4.a/3

{*AI12-0005-1*} {*AI12-0061-1*} {*AI12-0079-3*} {*AI12-0395-1*} {*AI12-0398-1*} basic_declaration 4.b/5

type_declaration
full_type_declaration
*type declaration syntax**
task_type_declaration*
protected_type_declaration*
incomplete_type_declaration -- NO
-- Incomplete types do not have aspects by definition,
-- , so it would not make sense to allow specifying them.
*private_type_declaration**
*private_extension_declaration**
*subtype_declaration**
object_declaration
*object declaration syntax**
*single_task_declaration**
*single_protected_declaration**
number_declaration -- NO
iterated_component_association -- NO
*subprogram_declaration**
*abstract_subprogram_declaration**
*null_procedure_declaration**
*expression_function_declaration**
package_declaration -- via* package_specification
*renaming_declaration**
-- There are no language-defined aspects that may be specified
-- on renames, but implementations might support some.
*exception_declaration**
generic_declaration
*generic_subprogram_declaration**
generic_package_declaration -- via* package_specification
*generic_instantiation**
enumeration_literal_specification -- NO
-- The syntax would be ambiguous if this was supported directly.
*discriminant_specification**
-- There are no language-defined aspects that may be specified
-- on discriminants, but implementations might support some.
*component_declaration**
loop_parameter_specification -- NO
chunk_specification -- NO - the enclosing construct has an aspect_specification
iterator_specification -- NO
iterator_parameter_specification -- NO
*parameter_specification**
-- There are no language-defined aspects that may be specified
-- on parameters, but implementations might support some.
subprogram_body -- - but language-defined aspects only if there is no explicit specification*
*entry_declaration**
*entry_index_specification**
-- There are no language-defined aspects that may be specified
-- on an entry index, but implementations might support some.
subprogram_body_stub -- - but language-defined aspects only if there is no explicit specification*
choice_parameter_specification -- NO
*generic_formal_parameter_declaration**
*formal_object_declaration**
formal_type_declaration
*formal_complete_type_declaration**
formal_incomplete_type_declaration -- NO - see incomplete_type_declaration *above*
formal_subprogram_declaration
*formal_concrete_subprogram_declaration**

```
   formal_abstract_subprogram_declaration*
  formal_package_declaration*
extended_return_object_declaration*
   -- There are no language-defined aspects that may be specified
   -- on return objects, but implementations might support some.
```

4.c/5 {*AI12-0005-1*} {*AI12-0169-1*}

```
 We also allow aspect_specifications on all kinds of bodies, but there are no language-defined aspects
-- that may be specified on a body. These are allowed for implementation-defined aspects.
-- See above for subprogram bodies and stubs (as these can be declarations).
package_body*
task_body*
protected_body*
entry_body*
package_body_stub*
task_body_stub*
protected_body_stub*
```

4.d/3 {*AI05-0267-1*} Syntactically, aspect_specifications generally are located at the end of declarations. When a declaration is all in one piece such as a null_procedure_declaration, object_declaration, or generic_instantiation the aspect_specification goes at the end of the declaration; it is then more visible and less likely to interfere with the layout of the rest of the structure. However, we make an exception for program units (other than subprogram specifications) and bodies, in which the aspect_specification goes before the **is**. In these cases, the entity could be large and could contain other declarations that also have aspect_specifications, so it is better to put the aspect_specification toward the top of the declaration. (Some aspects – such as Pure – also affect the legality of the contents of a unit, so it would be annoying to only see those after reading the entire unit.)

Name Resolution Rules

5/5 {*AI05-0183-1*} {*AI12-0212-1*} An aspect_mark identifies an aspect of the entity defined by the associated declaration (the *associated entity*); the aspect denotes an object, a value, an expression, an aggregate, a subprogram, or some other kind of entity. If the aspect_mark identifies:

6/3 • an aspect that denotes an object, the aspect_definition shall be a name. The expected type for the name is the type of the identified aspect of the associated entity;

7/3 • an aspect that is a value or an expression, the aspect_definition shall be an expression. The expected type for the expression is the type of the identified aspect of the associated entity;

7.1/5 • {*AI12-0212-1*} an aspect that is an aggregate, the aspect definition shall be an expression that is an aggregate, with the form of the aggregate determined by the identified aspect;

7.a/5 **Ramification:** {*AI12-0212-1*} The only requirement here is that the expression is syntactically an aggregate; there is no requirement that it resolve to some particular type or even that it should resolve like an aggregate. Each aspect that uses an aggregate is responsible for specifying how the choice(s), component(s), and other contents of the aggregate are resolved and interpreted.

8/3 • an aspect that denotes a subprogram, the aspect_definition shall be a name; the expected profile for the name is the profile required for the aspect of the associated entity;

9/3 • an aspect that denotes some other kind of entity, the aspect_definition shall be a name, and the name shall resolve to denote an entity of the appropriate kind;

10/3 • an aspect that is given by an identifier specific to the aspect, the aspect_definition shall be an identifier, and the identifier shall be one of the identifiers specific to the identified aspect.

11/5 {*AI05-0183-1*} {*AI12-0373-1*} {*AI12-0427-1*} The usage names in an aspect_definition associated with a declaration [are not resolved at the point of the associated declaration, but rather] are resolved at the end of the immediately enclosing declaration list, or in the case of the declaration of a library unit, at the end of the visible part of the entity.

11.a **Ramification:** If an aspect_specification is not associated with a declaration (instead being part of a statement or expression), then it is resolved when the associated construct is resolved.

{*AI05-0183-1*} {*AI12-0180-1*} {*AI12-0220-1*} If the associated declaration is for a subprogram, entry, or access-to-subprogram type, the names of the formal parameters are directly visible within the aspect_definition, as are certain attributes, as specified elsewhere in this document for the identified aspect. If the associated declaration is a type_declaration, within the aspect_definition the names of any visible components, protected subprograms, and entries are directly visible, and the name of the first subtype denotes the current instance of the type (see 8.6). If the associated declaration is a subtype_declaration, within the aspect_definition the name of the new subtype denotes the current instance of the subtype. 12/5

Legality Rules

{*AI05-0183-1*} If the first freezing point of the associated entity comes before the end of the immediately enclosing declaration list, then each usage name in the aspect_definition shall resolve to the same entity at the first freezing point as it does at the end of the immediately enclosing declaration list. 13/3

{*AI12-0181-1*} {*AI12-0396-1*} An expression or name that causes freezing of an entity shall not occur within an aspect_specification that specifies a representation or operational aspect of that entity. 13.1/5

Reason: {*AI12-0181-1*} {*AI12-0396-1*} The rule prevents an aspect of an entity from depending (directly or indirectly) on properties of the entity itself. 13.a/5

Implementation Note: {*AI12-0396-1*} This check needs to be deferred at least to the freezing of the entity (as the aspect_definition is not resolved until then), and might be accomplished during the freezing of the aspect_definition (since it is closely related). Keep in mind that multiple other entities could be involved. 13.b/5

{*AI05-0183-1*} At most one occurrence of each aspect_mark is allowed within a single aspect_specification. The aspect identified by the aspect_mark shall be an aspect that can be specified for the associated entity (or view of the entity defined by the associated declaration). 14/3

Ramification: {*AI12-0116-1*} {*AI12-0005-1*} {*AI12-0396-1*} This rule prevents multiple specifications in the same aspect_specification. Rules in 13.1 prevent multiple specifications in different aspect_specifications (on different views of the same type, for instance), or between operational or representation items and an aspect_specification. 14.a/5

{*AI05-0183-1*} The aspect_definition associated with a given aspect_mark may be omitted only when the aspect_mark identifies an aspect of a boolean type, in which case it is equivalent to the aspect_definition being specified as True. 15/3

{*AI05-0183-1*} If the aspect_mark includes 'Class, then the associated entity shall be a tagged type or a primitive subprogram of a tagged type. 16/3

{*AI05-0183-1*} {*AI05-0267-1*} {*AI12-0064-2*} {*AI12-0194-1*} {*AI12-0380-1*} {*AI12-0407-1*} Unless otherwise specified for a specific aspect, a language-defined aspect cannot be specified on a renaming_declaration or a generic_formal_parameter_declaration. 17/5

Discussion: {*AI12-0064-2*} Implementation-defined aspects can be allowed on these, of course; the implementation will need to define the semantics. In addition, the language does not define default aspect matching rules for generic formals; only the handful of aspects allowed on formals have such rules. Therefore, the implementation will need to define actual type matching rules for any aspects allowed on formal types. 17.a/5

{*AI05-0183-1*} {*AI05-0267-1*} {*AI12-0105-1*} {*AI12-0380-1*} Unless specified otherwise, a language-defined aspect shall not be specified in an aspect_specification given on a completion of a program unit. 18/5

Reason: Most language-defined aspects (for example, preconditions) are intended to be available to callers, and specifying them on a body that has a separate declaration hides them from callers. Specific language-defined aspects may allow this, but they have to do so explicitly (by defining an alternative Legality Rule), and provide any needed rules about visibility. Note that this rule does not apply to implementation-defined aspects, so implementers need to carefully define whether such aspects can be applied to bodies and stubs, and what happens if they are specified on both the declaration and body of a unit. 18.a/3

18.1/4 {*AI05-0183-1*} {*AI12-0138-1*} If an aspect of a derived type is inherited from an ancestor type and has the boolean value True, the inherited value shall not be overridden to have the value False for the derived type, unless otherwise specified in this document.

18.b/4 **Reason:** Most boolean-valued language-defined aspects are associated with a representation pragma. The existing rules for such pragmas assume that the aspect cannot be removed. For instance, if a type *T* is declared to be Atomic, then all descendants of *T* are also Atomic. This rule ensures that remains the case when using the aspect notation instead of pragmas.

18.b.1/5 **Discussion:** This definition leaves holes for Boolean aspects that can be specified on non-first subtypes. Such aspects (for instance, Nonblocking) must have their own rules (that is, "otherwise specify" rules) that define the effects of inheriting from subtypes (both first subtypes and nonfirst subtypes).

18.2/5 {*AI12-0396-1*} If a given aspect is type-related and inherited, then within an aspect_definition for the aspect, if a name resolves to denote multiple visible subprograms, all or none of the denoted subprograms shall be primitives of the associated type.

18.b.2/5 **Reason:** {*AI12-0396-1*} This is necessary because the inheritance rules for names denoting primitive subprograms are different from those for names denoting other entities — see 13.1.

18.3/5 {*AI12-0138-1*} {*AI12-0206-1*} {*AI12-0419-1*} Certain type-related aspects are defined to be *nonoverridable*; all such aspects are inherited by derived types according to the rules given in 13.1. Any legality rule associated with a nonoverridable aspect is re-checked for the derived type, if the derived type is not abstract. Certain type-related and subtype-specific aspects are defined to be *additive*; such aspects are not inherited, but they can *apply* to the types derived from, or the subtypes based on, the original type or subtype, as defined for each such aspect. Finally, certain type-related aspects are *implicitly composed*; such aspects are not inherited, but rather a default implementation for a derived type is provided, as defined for each such aspect, based on that of its parent type, presuming the aspect for the parent type is available where the derived type is declared, plus those of any new components added as part of a type extension.

18.4/5 {*AI12-0138-1*} {*AI12-0206-1*} {*AI12-0211-1*} {*AI12-0396-1*} If a nonoverridable aspect is directly specified for a type *T*, then any explicit specification of that aspect for any descendant of *T* (other than *T* itself) shall be *confirming*. In the case of an aspect that is a name, this means that the specified name shall *match* the inherited aspect in the sense that it shall denote the same declarations as would the inherited name. Similarly, for an aspect that is an expression or an aggregate, confirming means the defining expression is fully conformant (see 6.3.1) with the defining expression for the inherited aspect, with the added rule that an identifier that is specific to the aspect is the same as the corresponding identifier in the inherited aspect.

18.5/5 {*AI12-0138-1*} {*AI12-0396-1*} If a full type has a partial view, and a given nonoverridable aspect is allowed for both the full view and the partial view, then the given aspect for the partial view and the full view shall be the same: the aspect shall be directly specified only on the partial view; if the full type inherits the aspect, then a matching definition shall be specified (directly or by inheritance) for the partial view.

18.c/4 **Ramification:** In order to enforce these rules without breaking privacy, we cannot allow a private type that could have a particular overridable aspect to have a hidden definition of that aspect. There is no problem if the private type does not allow the aspect (as the aspect could not be specified on descendants in that case).

18.6/5 {*AI12-0211-1*} If a type inherits a nonoverridable aspect from multiple ancestors, the value of the aspect inherited from any given ancestor shall be confirming of the values inherited from all other ancestors.

18.d/5 **Reason:** {*AI12-0005-1*} {*AI12-0211-1*} If more than one progenitor of a type *T* specifies a nonoverridable aspect, they all have to specify the same or matching values for that aspect. Otherwise, we'd have two different values for the aspect that depend on which progenitor we inherit from.

18.7/5 {*AI12-0138-1*} {*AI12-0211-1*} In addition to the places where Legality Rules normally apply (see 12.3), these rules about nonoverridable aspects also apply in the private part of an instance of a generic unit.

Discussion: We don't need an assume-the-worst rule for most nonoverridable aspects as they only work on tagged types and deriving from formal tagged types is not allowed in generic bodies. In the case of Implicit_Dereference, a derivation in a generic body does not cause problems (the ancestor necessarily cannot have the aspect, else specifying the aspect would be illegal), as there could be no place with visibility on both aspects. In the case of Max_Entry_Queue_Length, it is only allowed on task and protected types, and on entries, and there are not formal versions of any of those things. In the case of No_Controlled_Parts, we defined an assume-the-worst rule with the aspect. 18.e/5

{*AI12-0437-1*} A list of all nonoverridable aspects can be found in the index, under “nonoverridable aspect”. 18.f/5

Static Semantics

{*AI05-0183-1*} Depending on which aspect is identified by the aspect_mark, an aspect_definition specifies: 19/3

- a name that denotes a subprogram, object, or other kind of entity; 20/3
- {*AI12-0427-1*} an expression (other than an aggregate), which is either evaluated to produce a single value, or which (as in a precondition) is to be evaluated at particular points during later execution; 21/5
- {*AI12-0427-1*} an identifier specific to the aspect; or 22/5
- {*AI12-0427-1*} an aggregate, which is positional or named, and is composed of elements of any of these four kinds of constructs. 22.1/5

{*AI05-0183-1*} The identified aspect of the associated entity, or in some cases, the view of the entity defined by the declaration, is as specified by the aspect_definition (or by the default of True when boolean). Whether an aspect_specification *applies* to an entity or only to the particular view of the entity defined by the declaration is determined by the aspect_mark and the kind of entity. The following aspects are view specific: 23/3

- An aspect specified on an object_declaration; 24/3
- An aspect specified on a subprogram_declaration; 25/3
- An aspect specified on a renaming_declaration. 26/3

{*AI05-0183-1*} All other aspect_specifications are associated with the entity, and *apply* to all views of the entity, unless otherwise specified in this document. 27/3

{*AI05-0183-1*} {*AI12-0106-1*} If the aspect_mark includes 'Class (a *class-wide aspect*), then, unless specified otherwise for a particular class-wide aspect: 28/4

- if the associated entity is a tagged type, the specification *applies* to all descendants of the type; 29/3
- if the associated entity is a primitive subprogram of a tagged type *T*, the specification *applies* to the corresponding primitive subprogram of all descendants of *T*. 30/3

{*AI05-0183-1*} {*AI05-0229-1*} All specifiable operational and representation attributes may be specified with an aspect_specification instead of an attribute_definition_clause (see 13.3). 31/3

Ramification: The name of the aspect is the same as that of the attribute (see 13.3), so the aspect_mark is the attribute_designator of the attribute. 31.a/3

{*AI12-0005-1*} Unless specified otherwise, all of the requirements for specifying a particular aspect with an attribute_definition_clause also apply to an aspect_specification for the aspect. These are enforced at the freezing point of the entity. For example, when specifying the Size aspect of a subtype, the expression has to be a static expression with an integer type and a nonnegative value, all of the recommended level of support requirements apply if Annex C is supported (see C.2), and so on. 31.b/5

{*AI05-0229-1*} {*AI12-0154-1*} {*AI12-0417-1*} Some aspects are defined to be *library unit aspects*. Library unit aspects are of type Boolean. The expression specifying a library unit aspect shall be static. Library unit aspects are defined for all program units, but shall be specified only for library units. Notwithstanding what this document says elsewhere, the expression of a library unit aspect is resolved 32/5

and evaluated at the point where it occurs in the aspect_specification[, rather than the first freezing point of the associated unit].

32.a/3 **Ramification:** The name of the aspect is the same as that of the pragma (see 13.1), so the aspect_mark is the name of the pragma.

33/3 {*AI05-0229-1*} In addition, other operational and representation aspects not associated with specifiable attributes or representation pragmas may be specified, as specified elsewhere in this document.

34/4 *This paragraph was deleted.*{*AI05-0183-1*} {*AI12-0138-1*}

35/3 {*AI05-0183-1*} If a Legality Rule or Static Semantics rule only applies when a particular aspect has been specified, the aspect is considered to have been specified only when the aspect_specification or attribute_definition_clause is visible (see 8.3) at the point of the application of the rule.

35.a/3 **Reason:** Some rules only apply when an aspect has been specified (for instance, an indexable type is one that has aspect Variable_Indexing specified). In order to prevent privacy breaking, this can only be true when the specification of the aspect is visible. In particular, if the Variable_Indexing aspect is specified on the full view of a private type, the private type is not considered an indexable type.

36/3 {*AI05-0183-1*} Alternative legality and semantics rules may apply for particular aspects, as specified elsewhere in this document.

Dynamic Semantics

37/5 {*AI05-0183-1*} {*AI12-0355-2*} At the freezing point of the associated entity, the aspect_specification is elaborated. When appearing in a construct other than a declaration, an aspect_specification is elaborated as part of the execution of the construct. The elaboration of the aspect_specification consists of the elaboration of each aspect_definition in an arbitrary order. The elaboration of an aspect_definition includes the evaluation of any name or expression that is part of the aspect_definition unless the part is itself an expression. If the corresponding aspect (or part thereof) represents an expression (as in a precondition), the elaboration of that part has no effect; the expression is evaluated later at points within the execution as specified elsewhere in this document for the particular aspect.

Implementation Permissions

38/3 {*AI05-0183-1*} Implementations may support implementation-defined aspects. The aspect_specification for an implementation-defined aspect may use an implementation-defined syntax for the aspect_definition, and may follow implementation-defined legality and semantics rules.

38.a/5 **Discussion:** {*AI12-0389-1*} The intent is to allow implementations to support aspects that are defined, for example, by a subtype_indication rather than an expression or a name. We chose not to try to enumerate all possible aspect_definition syntaxes, but to give implementations maximum freedom.

38.a.1/3 **Implementation defined:** Implementation-defined aspects, including the syntax for specifying such aspects and the legality rules for such aspects.

39/5 {*AI12-0389-1*} An implementation may ignore the specification of an unrecognized aspect; if an implementation chooses to ignore such an aspect specification (as opposed to rejecting it), then it has no effect on the semantics of the program except for possibly (and this is not required) the rejection of syntax errors within the aspect_definition.

39.a/5 **Discussion:** {*AI12-0389-1*} Identifying the textual end of an aspect_definition for an unrecognized aspect may be challenging, particularly if the syntax for the unrecognized aspect's aspect_definition is implementation-defined. It is not specified how an implementation might accomplish this. Note that an implementation is never *required* to be able to do this; if an aspect_definition for an unrecognized aspect is problematic in any way, then it can always be rejected (as opposed to being ignored).

Extensions to Ada 2005

39.b/3 {*AI05-0183-1*} {*AI05-0229-1*} {*AI05-0267-1*} Aspect specifications are new.

Inconsistencies With Ada 2012

{*AI12-0180-1*} **Correction:** Names of protected subprograms and entries are now directly visible in an aspect of a type declaration. This was always intended to be the case, but it was omitted from the Reference Manual by an editing error. In the unlikely case that a parameterless protected function has the same name and type as an entity used in a type invariant expression, the meaning would change from the outside entity which would now be hidden by the protected function. It is much more likely that a conflict (itself rather unlikely) would cause a resolution failure. 39.c/5

Incompatibilities With Ada 2012

{*AI12-0154-1*} **Corrigendum:** Added a clarification that aspects that correspond to library unit pragmas are resolved and evaluated immediately. This is incompatible, as a reference to an entity defined after the aspect will now be illegal. However, this would have required retroactive enforcement of such aspects, which is a new capability not available from the associated pragma, and moreover no known Ada 2012 implementation has ever allowed late evaluation of such aspects. As such, there should be no practical incompatibility. 39.d/4

{*AI12-0005-1*} {*AI12-0220-1*} Parameters of access-to-subprogram types are now visible in aspect specifications. This can be incompatible if some entity with the same name as a parameter is used in an existing aspect specification. We believe that all such aspects require either static expressions or a subprogram that is statically denoted; since a parameter can be neither of these and hides everything else, all such cases will be caught at compile-time. In addition, we expect such cases to be very rare. 39.e/5

Extensions to Ada 2012

{*AI12-0212-1*} An aspect can be an unresolved aggregate, these can be used to specify a list of entities, or to set a group of different but related properties with a single aspect. 39.f/5

{*AI12-0355-2*} Aspect specifications can appear in, and apply to, constructs other than declarations. 39.g/5

{*AI12-0389-1*} An unrecognized aspect specification can be ignored. 39.h/5

Wording Changes from Ada 2012

{*AI12-0105-1*} **Corrigendum:** Clarified the wording so that the restriction against language-defined aspects on subprogram completions includes completions that are expressions functions and null procedures. 39.i/4

{*AI12-0106-1*} **Corrigendum:** Defined class-wide aspect for use in rules in 13.13.2. 39.j/4

{*AI12-0138-1*} **Corrigendum:** Added a definition of nonoverridable aspects. This is necessary to prevent generic contract problems with formal derived types. 39.k/4

{*AI12-0194-1*} **Correction:** Added entry_body to the list of entities that don't allow any language-defined aspects. This was an oversight in AI12-0169-1 which allowed aspect_specifications on an entry_body in the first place. 39.l/5

{*AI12-0206-1*} **Correction:** Extended the definition of nonoverridable aspects to cover most kinds of aspects. 39.m/5

{*AI12-0211-1*} **Correction:** Added a rule so that the a nonoverridable aspect has to be the same for every ancestor of a type. 39.n/5

{*AI12-0355-2*} {*AI12-0427-1*} Clarified when elaboration and resolution of aspect_specifications not associated with a declaration occur. 39.o/5

{*AI12-0380-1*} {*AI12-0407-1*} Revised the rules about entities that cannot have specified language-defined aspects. The body portion of the rule was changed to a general prohibition on specifying language-defined aspects on completions (this allows specifying aspects on bodies that act as a declaration, and eliminates the list of entities, which was a maintenance nightmare). Then the remaining entities (renamings and generic formal parameters) were revised to allow specific aspects, since Nonblocking (see 9.5), as well as Pre and Post, are now allowed on generic formal parameters. 39.p/5

{*AI12-0373-1*} Added the missing definition for resolution of entities found in aspects of library units. 39.q/5

{*AI12-0396-1*} **Correction:** Defined confirming for nonoverridable aspects that are expressions or aggregates. 39.r/5

{*AI12-0419-1*} Added a description of additive and implicitly composed aspects in order to explain the different inheritance mechanisms used by existing aspects. 39.s/5

13.2 Packed Types

{*AI05-0229-1*} [The Pack aspect having the value True specifies that storage minimization should be the main criterion when selecting the representation of a composite type.] 1/3

Paragraphs 2 through 4 were moved to Annex J, "Obsolescent Features".

Language Design Principles

1.a/4 {*AI12-0001-1*} If the default representation already uses minimal storage for a particular type, aspect Pack might not cause any representation change. It follows that aspect Pack should always be allowed, even when it has no effect on representation.

1.b/4 As a consequence, the chosen representation for a packed type may change during program maintenance even if the type is unchanged (in particular, if other representation aspects change on a part of the type). This is different than the behavior of most other representation aspects, whose properties remain guaranteed no matter what changes are made to other aspects.

1.c/4 Therefore, aspect Pack should not be used to achieve a representation required by external criteria. For instance, setting Component_Size to 1 should be preferred over using aspect Pack to ensure an array of bits. If future maintenance would make the array components aliased, independent, or atomic, the program would become illegal if Component_Size is used (immediately identifying a problem) while the aspect Pack version would simply change representations (probably causing a hard-to-find bug).

Static Semantics

5/3 {*AI05-0229-1*} For a full type declaration of a composite type, the following language-defined representation aspect may be specified:

5.1/3 Pack The type of aspect Pack is Boolean. When aspect Pack is True for a type, the type (or the extension part) is said to be *packed*. For a type extension, the parent part is packed as for the parent type, and specifying Pack causes packing only of the extension part.

5.a/3 **Aspect Description for Pack:** Minimize storage when laying out records and arrays.

5.2/3 If directly specified, the aspect_definition shall be a static expression. If not specified (including by inheritance), the aspect is False.

5.b/3 **Ramification:** {*AI05-0229-1*} The only high level semantic effect of specifying the Pack aspect is potential loss of independent addressability (see 9.10, “Shared Variables”).]

Implementation Advice

6 If a type is packed, then the implementation should try to minimize storage allocated to objects of the type, possibly at the expense of speed of accessing components, subject to reasonable complexity in addressing calculations.

6.a.1/2 **Implementation Advice:** Storage allocated to objects of a packed type should be minimized.

6.a/3 **Ramification:** {*AI05-0229-1*} Specifying the Pack aspect is for gaining space efficiency, possibly at the expense of time. If more explicit control over representation is desired, then a record_representation_clause, a Component_Size clause, or a Size clause should be used instead of, or in addition to, the Pack aspect.

6.1/4 *This paragraph was deleted.*{*AI95-00291-02*} {*AI12-0001-1*}

7/3 {*AI05-0229-1*} The recommended level of support for the Pack aspect is:

7.1/4 • {*AI12-0001-1*} Any component of a packed type that is of a by-reference type, that is specified as independently addressable, or that contains an aliased part, shall be aligned according to the alignment of its subtype.

7.a/4 **Ramification:** This also applies to atomic components. "Atomic" implies "specified as independently addressable", so we don't need to mention atomic here.

7.b/4 Other components do not have to respect the alignment of the subtype when packed; in many cases, the Recommended Level of Support will require the alignment to be ignored.

8/5 • {*AI12-0001-1*} {*AI12-0444-1*} For a packed record type, the components should be packed as tightly as possible subject to the above alignment requirements, the Sizes of the component subtypes, and any record_representation_clause that applies to the type; the implementation is allowed to reorder components or cross aligned word boundaries to improve the packing. A component whose Size is greater than the word size may be allocated an integral number of words.

Ramification: The implementation can always allocate an integral number of words for a component that will not fit in a word. The rule also allows small component sizes to be rounded up if such rounding does not waste space. For example, if Storage_Unit = 8, then a component of size 8 is probably more efficient than a component of size 7 plus a 1-bit gap (assuming the gap is needed anyway). 8.a

- {*AI05-0009-1*} {*AI12-0001-1*} For a packed array type, if the Size of the component subtype is less than or equal to the word size, Component_Size should be less than or equal to the Size of the component subtype, rounded up to the nearest factor of the word size, unless this would violate the above alignment requirements. 9/4

This paragraph was deleted.{*AI12-0001-1*} 9.a/4

Implementation Advice: The recommended level of support for the Pack aspect should be followed. 9.b/3

Wording Changes from Ada 95

{*AI95-00291-02*} {*AI05-0229-1*} Added clarification that the Pack aspect can ignore alignment requirements on types that don't have by-reference or aliased parts. This was always intended, but there was no wording to that effect. 9.c/3

Extensions to Ada 2005

{*AI05-0229-1*} Aspect Pack is new; pragma Pack is now obsolescent. 9.d/3

Wording Changes from Ada 2005

{*AI05-0009-1*} **Correction:** Fixed so that the presence or absence of a confirming Component_Size representation clause does not change the meaning of the Pack aspect. 9.e/3

Wording Changes from Ada 2012

{*AI12-0001-1*} **Corrigendum:** Fixed so that the Recommended Level of Support does not require packing of components for which such packing would violate other representation items or aspects. This is not incompatible, as either such Pack aspects were treated as illegal or the Recommended Level of Support was ignored as impractical, neither of which would change the behavior of any working programs. (Other behavior cannot be justifed from the Reference Manual.) 9.f/4

13.3 Operational and Representation Attributes

{*8652/0009*} {*AI95-00137-01*} [The values of certain implementation-dependent characteristics can be obtained by interrogating appropriate operational or representation attributes. Some of these attributes are specifiable via an attribute_definition_clause.] 1/1

Language Design Principles

In general, the meaning of a given attribute should not depend on whether the attribute was specified via an attribute_definition_clause, or chosen by default by the implementation. 1.a

Syntax

```
attribute_definition_clause ::=
    for local_name'attribute_designator use expression;
  | for local_name'attribute_designator use name;
```
2

Name Resolution Rules

For an attribute_definition_clause that specifies an attribute that denotes a value, the form with an expression shall be used. Otherwise, the form with a name shall be used. 3

For an attribute_definition_clause that specifies an attribute that denotes a value or an object, the expected type for the expression or name is that of the attribute. For an attribute_definition_clause that specifies an attribute that denotes a subprogram, the expected profile for the name is the profile required for the attribute. For an attribute_definition_clause that specifies an attribute that denotes some other kind of entity, the name shall resolve to denote an entity of the appropriate kind. 4

4.a **Ramification:** For example, the Size attribute is of type *universal_integer*. Therefore, the expected type for Y in "**for** X'Size **use** Y;" is *universal_integer*, which means that Y can be of any integer type.

4.b **Discussion:** For attributes that denote subprograms, the required profile is indicated separately for the individual attributes.

4.c **Ramification:** For an attribute_definition_clause with a name, the name need not statically denote the entity it denotes. For example, the following kinds of things are allowed:

4.d
```
for Some_Access_Type'Storage_Pool use Storage_Pool_Array(I);
for Some_Type'Read use Subprogram_Pointer.all;
```

Legality Rules

5/3 {*8652/0009*} {*AI95-00137-01*} {*AI05-0183-1*} An attribute_designator is allowed in an attribute_definition_clause only if this Reference Manual explicitly allows it, or for an implementation-defined attribute if the implementation allows it. Each specifiable attribute constitutes an operational aspect or aspect of representation; the name of the aspect is that of the attribute.

5.a **Discussion:** For each specifiable attribute, we generally say something like, "The ... attribute may be specified for ... via an attribute_definition_clause."

5.b The above wording allows for T'Class'Alignment, T'Class'Size, T'Class'Input, and T'Class'Output to be specifiable.

5.c A specifiable attribute is not necessarily specifiable for all entities for which it is defined. For example, one is allowed to ask T'Component_Size for an array subtype T, but "**for** T'Component_Size **use** ..." is only allowed if T is a first subtype, because Component_Size is a type-related aspect.

6/5 *This paragraph was deleted.*{*AI12-0427-1*}

6.a/5 *This paragraph was deleted.*

6.b/5 `This paragraph was deleted.`

6.c/5 *This paragraph was deleted.*

Static Semantics

7/2 {*AI95-00270-01*} A *Size clause* is an attribute_definition_clause whose attribute_designator is Size. Similar definitions apply to the other specifiable attributes.

7.a **To be honest:** An attribute_definition_clause is type-related or subtype-specific if the attribute_designator denotes a type-related or subtype-specific attribute, respectively.

8 A *storage element* is an addressable element of storage in the machine. A *word* is the largest amount of storage that can be conveniently and efficiently manipulated by the hardware, given the implementation's run-time model. A word consists of an integral number of storage elements.

8.a **Discussion:** A storage element is not intended to be a single bit, unless the machine can efficiently address individual bits.

8.b **Ramification:** For example, on a machine with 8-bit storage elements, if there exist 32-bit integer registers, with a full set of arithmetic and logical instructions to manipulate those registers, a word ought to be 4 storage elements — that is, 32 bits.

8.c **Discussion:** The "given the implementation's run-time model" part is intended to imply that, for example, on an 80386 running MS-DOS, the word might be 16 bits, even though the hardware can support 32 bits.

8.d A word is what ACID refers to as a "natural hardware boundary".

8.e Storage elements may, but need not be, independently addressable (see 9.10, "Shared Variables"). Words are expected to be independently addressable.

8.1/3 {*AI95-00133-01*} {*AI05-0092-1*} A *machine scalar* is an amount of storage that can be conveniently and efficiently loaded, stored, or operated upon by the hardware. Machine scalars consist of an integral number of storage elements. The set of machine scalars is implementation defined, but includes at least the storage element and the word. Machine scalars are used to interpret component_clauses when the nondefault bit ordering applies.

8.e.1/2 **Implementation defined:** The set of machine scalars.

Ramification: {*AI05-0092-1*} A single storage element is a machine scalar in all Ada implementations. Similarly, a word is a machine scalar in all implementations (although it might be the same as a storage element). An implementation may define other machine scalars that make sense on the target (a half-word, for instance). 8.f/3

{*8652/0009*} {*AI95-00137-01*} {*AI05-0191-1*} {*AI12-0059-1*} The following representation attributes are defined: Address, Alignment, Size, Object_Size, Storage_Size, Component_Size, Has_Same_Storage, and Overlaps_Storage. 9/5

For a prefix X that denotes an object, program unit, or label: 10/1

X'Address Denotes the address of the first of the storage elements allocated to X. For a program unit or label, this value refers to the machine code associated with the corresponding body or statement. The value of this attribute is of type System.Address. 11

Ramification: Here, the "first of the storage elements" is intended to mean the one with the lowest address; the endianness of the machine doesn't matter. 11.a

{*AI05-0095-1*} The prefix of X'Address shall not statically denote a subprogram that has convention Intrinsic. X'Address raises Program_Error if X denotes a subprogram that has convention Intrinsic. 11.1/3

Address may be specified for stand-alone objects and for program units via an attribute_definition_clause. 12

Ramification: Address is not allowed for enumeration literals, predefined operators, derived task types, or derived protected types, since they are not program units. 12.a

Address is not allowed for intrinsic subprograms, either. That can be checked statically unless the prefix is a generic formal subprogram and the attribute reference is in the body of a generic unit. We define that case to raise Program_Error, in order that the compiler does not have to build a wrapper for intrinsic subprograms. 12.b/3

The validity of a given address depends on the run-time model; thus, in order to use Address clauses correctly, one needs intimate knowledge of the run-time model. 12.c

{*AI05-0229-1*} If the Address of an object is specified, any explicit or implicit initialization takes place as usual, unless the Import aspect is also specified for the object (in which case any necessary initialization is presumably done in the foreign language). 12.d/3

Any compilation unit containing an attribute_reference of a given type depends semantically on the declaration of the package in which the type is declared, even if not mentioned in an applicable with_clause — see 10.1.1. In this case, it means that if a compilation unit contains X'Address, then it depends on the declaration of System. Otherwise, the fact that the value of Address is of a type in System wouldn't make sense; it would violate the "legality determinable via semantic dependences" Language Design Principle. 12.e

AI83-00305 — If X is a task type, then within the body of X, X denotes the current task object; thus, X'Address denotes the object's address. 12.f

Interrupt entries and their addresses are described in J.7.1, "Interrupt Entries". 12.g

If X is not allocated on a storage element boundary, X'Address points at the first of the storage elements that contains any part of X. This is important for the definition of the Position attribute to be sensible. 12.h

Aspect Description for Address: Machine address of an entity. 12.i/3

Erroneous Execution

{*AI05-0009-1*} If an Address is specified, it is the programmer's responsibility to ensure that the address is valid and appropriate for the entity and its use; otherwise, program execution is erroneous. 13/3

Discussion: "Appropriate for the entity and its use" covers cases such as misaligned addresses, read-only code addresses for variable data objects (and nonexecutable data addresses for code units), and addresses which would force objects that are supposed to be independently addressable to not be. Such addresses may be "valid" as they designate locations that are accessible to the program, but the program execution is still erroneous (meaning that implementations do not have to worry about these cases). 13.a

Implementation Advice

For an array X, X'Address should point at the first component of the array, and not at the array bounds. 14

14.a.1/2 **Implementation Advice:** For an array X, X'Address should point at the first component of the array rather than the array bounds.

14.a **Ramification:** On the other hand, we have no advice to offer about discriminants and tag fields; whether or not the address points at them is not specified by the language. If discriminants are stored separately, then the Position of a discriminant might be negative, or might raise an exception.

15 The recommended level of support for the Address attribute is:

16 • X'Address should produce a useful result if X is an object that is aliased or of a by-reference type, or is an entity whose Address has been specified.

16.a **Reason:** Aliased objects are the ones for which the Unchecked_Access attribute is allowed; hence, these have to be allocated on an addressable boundary anyway. Similar considerations apply to objects of a by-reference type.

16.b An implementation need not go to any trouble to make Address work in other cases. For example, if an object X is not aliased and not of a by-reference type, and the implementation chooses to store it in a register, X'Address might return System.Null_Address (assuming registers are not addressable). For a subprogram whose calling convention is Intrinsic, or for a package, the implementation need not generate an out-of-line piece of code for it.

17 • An implementation should support Address clauses for imported subprograms.

18/2 • *This paragraph was deleted.*{*AI95-00291-02*}

18.a/2 *This paragraph was deleted.*

19 • If the Address of an object is specified, or it is imported or exported, then the implementation should not perform optimizations based on assumptions of no aliases.

19.a/2 **Implementation Advice:** The recommended level of support for the Address attribute should be followed.

20 NOTE 1 The specification of a link name with the Link_Name aspect (see B.1) for a subprogram or object is an alternative to explicit specification of its link-time address, allowing a link-time directive to place the subprogram or object within memory.

21 NOTE 2 The rules for the Size attribute imply, for an aliased object X, that if X'Size = Storage_Unit, then X'Address points at a storage element containing all of the bits of X, and only the bits of X.

Wording Changes from Ada 83

21.a The intended meaning of the various attributes, and their attribute_definition_clauses, is more explicit.

21.b The address_clause has been renamed to at_clause and moved to Annex J, “Obsolescent Features”. One can use an Address clause (“for T'Address **use** ...;”) instead.

21.c The attributes defined in RM83-13.7.3 are moved to Annex G, A.5.3, and A.5.4.

Wording Changes from Ada 2005

21.c.1/3 {*AI05-0183-1*} Defined that the names of aspects are the same as the name of the attribute; that gives a name to use in aspect_specifications (see 13.1.1).

Language Design Principles

21.d By default, the Alignment of a subtype should reflect the “natural” alignment for objects of the subtype on the machine. The Alignment, whether specified or default, should be known at compile time, even though Addresses are generally not known at compile time. (The generated code should never need to check at run time the number of zero bits at the end of an address to determine an alignment).

21.e There are two symmetric purposes of Alignment clauses, depending on whether or not the implementation has control over object allocation. If the implementation allocates an object, the implementation should ensure that the Address and Alignment are consistent with each other. If something outside the implementation allocates an object, the implementation should be allowed to assume that the Address and Alignment are consistent, but should not assume stricter alignments than that.

Static Semantics

22/2 {*AI95-00291-02*} For a prefix X that denotes an object:

23/2 X'Alignment

{*AI95-00291-02*} The value of this attribute is of type *universal_integer*, and nonnegative; zero means that the object is not necessarily aligned on a storage element boundary. If

X'Alignment is not zero, then X is aligned on a storage unit boundary and X'Address is an integral multiple of X'Alignment (that is, the Address modulo the Alignment is zero).

This paragraph was deleted.{*AI95-00291-02*} 24/2

Ramification: The Alignment is passed by an allocator to the Allocate operation; the implementation has to choose a value such that if the address returned by Allocate is aligned as requested, the generated code can correctly access the object. 24.a

The above mention of "modulo" is referring to the "**mod**" operator declared in System.Storage_Elements; if X **mod** N = 0, then X is by definition aligned on an N-storage-element boundary. 24.b

{*AI95-00291-02*} Alignment may be specified for [stand-alone] objects via an attribute_definition_clause; the expression of such a clause shall be static, and its value nonnegative. 25/2

Aspect Description for Alignment (object): Alignment of an object. 25.a/3

This paragraph was deleted.{*AI95-00247-01*} 26/2

{*AI95-00291-02*} For every subtype S: 26.1/2

S'Alignment 26.2/2

{*AI95-00291-02*} The value of this attribute is of type *universal_integer*, and nonnegative.

{*AI95-00051-02*} {*AI95-00291-02*} For an object X of subtype S, if S'Alignment is not zero, then X'Alignment is a nonzero integral multiple of S'Alignment unless specified otherwise by a representation item. 26.3/2

{*AI95-00291-02*} Alignment may be specified for first subtypes via an attribute_definition_clause; the expression of such a clause shall be static, and its value nonnegative. 26.4/2

Aspect Description for Alignment (subtype): Alignment of a subtype. 26.a/3

Erroneous Execution

Program execution is erroneous if an Address clause is given that conflicts with the Alignment. 27

Ramification: The user has to either give an Alignment clause also, or else know what Alignment the implementation will choose by default. 27.a

{*AI95-00051-02*} {*AI95-00291-02*} For an object that is not allocated under control of the implementation, execution is erroneous if the object is not aligned according to its Alignment. 28/2

Implementation Advice

{*AI05-0116-1*} For any tagged specific subtype *S*, *S*'Class'Alignment should equal *S*'Alignment. 28.1/3

Reason: A tagged object should never be less aligned than the alignment of the type of its view, so for a class-wide type T'Class, the alignment should be no greater than that of any type covered by T'Class. If the implementation only supports alignments that are required by the recommended level of support (and this is most likely), then the alignment of any covered type has to be the same or greater than that of T — which leaves the only reasonable value of T'Class'Alignment being T'Alignment. Thus we recommend this, but don't require it so that in the unlikely case that the implementation does support smaller alignments for covered types, it can select a smaller value for T'Class'Alignment. 28.a/3

Implementation Advice: For any tagged specific subtype *S*, *S*'Class'Alignment should equal *S*'Alignment. 28.a.1/3

The recommended level of support for the Alignment attribute for subtypes is: 29

- {*AI95-00051-02*} An implementation should support an Alignment clause for a discrete type, fixed point type, record type, or array type, specifying an Alignment value that is zero or a power of two, subject to the following: 30/2
- {*AI95-00051-02*} {*AI12-0444-1*} An implementation is not required to support an Alignment clause for a signed integer type specifying an Alignment greater than the largest Alignment value that is ever chosen by default by the implementation for any signed integer type. A corresponding limitation may be imposed for modular integer types, fixed point types, enumeration types, record types, and array types. 31/5

32/5 • {*AI95-00051-02*} {*AI12-0444-1*} An implementation is not required to support a nonconfirming Alignment clause that can cause the creation of an object of an elementary type that cannot be easily loaded and stored by available machine instructions.

32.1/5 • {*AI95-00291-02*} {*AI12-0444-1*} An implementation is not required to support an Alignment specified for a derived tagged type that is not a multiple of the Alignment of the parent type. An implementation is not required to support a nonconfirming Alignment specified for a derived untagged by-reference type.

32.a/2 **Ramification:** {*AI95-00291-02*} There is no recommendation to support any nonconfirming Alignment clauses for types not mentioned above. Remember that 13.1 requires support for confirming Alignment clauses for all types.

32.b/3 **Implementation Note:** {*AI05-0116-1*} An implementation that tries to support other alignments for derived tagged types will need to allow inherited subprograms to be passed objects that are less aligned than expected by the parent subprogram and type. This is unlikely to work if alignment has any effect on code selection. Similar issues arise for untagged derived types whose parameters are passed by reference.

33 The recommended level of support for the Alignment attribute for objects is:

34/2 • *This paragraph was deleted.*{*AI95-00291-02*}

35 • For stand-alone library-level objects of statically constrained subtypes, the implementation should support all Alignments supported by the target linker. For example, page alignment is likely to be supported for such objects, but not for subtypes.

35.1/2 • {*AI95-00291-02*} For other objects, an implementation should at least support the alignments supported for their subtype, subject to the following:

35.2/5 • {*AI95-00291-02*} {*AI12-0444-1*} An implementation is not required to support Alignments specified for objects of a by-reference type or for objects of types containing aliased subcomponents if the specified Alignment is not a multiple of the Alignment of the subtype of the object.

35.a/2 **Implementation Advice:** The recommended level of support for the Alignment attribute should be followed.

36 NOTE 3 Alignment is a subtype-specific attribute.

37/2 *This paragraph was deleted.*{*AI95-00247-01*}

37.a/2 *This paragraph was deleted.*

38/3 NOTE 4 {*AI05-0229-1*} {*AI05-0269-1*} A component_clause, Component_Size clause, or specifying the Pack aspect as True can override a specified Alignment.

38.a **Discussion:** Most objects are allocated by the implementation; for these, the implementation obeys the Alignment. The implementation is of course allowed to make an object *more* aligned than its Alignment requires — an object whose Alignment is 4 might just happen to land at an address that's a multiple of 4096. For formal parameters, the implementation might want to force an Alignment stricter than the parameter's subtype. For example, on some systems, it is customary to always align parameters to 4 storage elements.

38.b Hence, one might initially assume that the implementation could evilly make all Alignments 1 by default, even though integers, say, are normally aligned on a 4-storage-element boundary. However, the implementation cannot get away with that — if the Alignment is 1, the generated code cannot assume an Alignment of 4, at least not for objects allocated outside the control of the implementation.

38.c Of course implementations can assume anything they can prove, but typically an implementation will be unable to prove much about the alignment of, say, an imported object. Furthermore, the information about where an address "came from" can be lost to the compiler due to separate compilation.

38.d/3 {*AI95-00114-01*} {*AI05-0229-1*} The Alignment of an object that is a component of a packed composite object will usually be 0, to indicate that the component is not necessarily aligned on a storage element boundary. For a subtype, an Alignment of 0 means that objects of the subtype are not normally aligned on a storage element boundary at all. For example, an implementation might choose to make Component_Size be 1 for an array of Booleans, even when the Pack aspect has not been specified for the array. In this case, Boolean'Alignment would be 0. (In the presence of tasking, this would in general be feasible only on a machine that had atomic test-bit and set-bit instructions.)

38.e If the machine has no particular natural alignments, then all subtype Alignments will probably be 1 by default.

38.f Specifying an Alignment of 0 in an attribute_definition_clause does not require the implementation to do anything (except return 0 when the Alignment is queried). However, it might be taken as advice on some implementations.

It is an error for an Address clause to disobey the object's Alignment. The error cannot be detected at compile time, in general, because the Address is not necessarily known at compile time (and is almost certainly not static). We do not require a runtime check, since efficiency seems paramount here, and Address clauses are treading on thin ice anyway. Hence, this misuse of Address clauses is just like any other misuse of Address clauses — it's erroneous. 38.g

A type extension can have a stricter Alignment than its parent. This can happen, for example, if the Alignment of the parent is 4, but the extension contains a component with Alignment 8. The Alignment of a class-wide type or object will have to be the maximum possible Alignment of any extension. 38.h

The recommended level of support for the Alignment attribute is intended to reflect a minimum useful set of capabilities. An implementation can assume that all Alignments are multiples of each other — 1, 2, 4, and 8 might be the only supported Alignments for subtypes. An Alignment of 3 or 6 is unlikely to be useful. For objects that can be allocated statically, we recommend that the implementation support larger alignments, such as 4096. We do not recommend such large alignments for subtypes, because the maximum subtype alignment will also have to be used as the alignment of stack frames, heap objects, and class-wide objects. Similarly, we do not recommend such large alignments for stack-allocated objects. 38.i

If the maximum default Alignment is 8 (say, Long_Float'Alignment = 8), then the implementation can refuse to accept stricter alignments for subtypes. This simplifies the generated code, since the compiler can align the stack and class-wide types to this maximum without a substantial waste of space (or time). 38.j

Note that the recommended level of support takes into account interactions between Size and Alignment. For example, on a 32-bit machine with 8-bit storage elements, where load and store instructions have to be aligned according to the size of the thing being loaded or stored, the implementation might accept an Alignment of 1 if the Size is 8, but might reject an Alignment of 1 if the Size is 32. On a machine where unaligned loads and stores are merely inefficient (as opposed to causing hardware traps), we would expect an Alignment of 1 to be supported for any Size. 38.k

Wording Changes from Ada 83

{*AI12-0005-1*} The nonnegative part is missing from RM83 (for mod_clauses, known in Ada 83 as alignment_clauses, which are an obsolete version of Alignment clauses). 38.l/5

Static Semantics

For a prefix X that denotes an object: 39/1

X'Size Denotes the size in bits of the representation of the object. The value of this attribute is of the type *universal_integer*. 40

Ramification: Note that Size is in bits even if Machine_Radix is 10. Each decimal digit (and the sign) is presumably represented as some number of bits. 40.a

Size may be specified for [stand-alone] objects via an attribute_definition_clause; the expression of such a clause shall be static and its value nonnegative. 41

Aspect Description for Size (object): Size in bits of an object. 41.a/3

Implementation Advice

{*AI95-00051-02*} The size of an array object should not include its bounds. 41.1/2

Implementation Advice: The Size of an array object should not include its bounds. 41.a.1/2

{*AI95-00051-02*} {*AI95-00291-02*} {*AI12-0445-1*} The recommended level of support for the Size attribute of objects is the same as for subtypes (see below), except that only a confirming Size clause is required to be supported for an aliased elementary object. 42/5

- *This paragraph was deleted.*{*AI95-00051-02*} 43/2

Static Semantics

For every subtype S: 44

S'Size If S is definite, denotes the size [(in bits)] that the implementation would choose for the following objects of subtype S: 45

- A record component of subtype S when the record type is packed. 46

47 - The formal parameter of an instance of Unchecked_Conversion that converts from subtype S to some other subtype.

48 If S is indefinite, the meaning is implementation defined. The value of this attribute is of the type *universal_integer*. The Size of an object is at least as large as that of its subtype, unless the object's Size is determined by a Size clause, a component_clause, or a Component_Size clause. Size may be specified for first subtypes via an attribute_definition_clause; the expression of such a clause shall be static and its value nonnegative.

48.a **Implementation defined:** The meaning of Size for indefinite subtypes.

48.b **Reason:** The effects of specifying the Size of a subtype are:

48.c - Unchecked_Conversion works in a predictable manner.

48.d - A composite type cannot be packed so tightly as to override the specified Size of a component's subtype.

48.e - Assuming the Implementation Advice is obeyed, if the specified Size allows independent addressability, then the Size of certain objects of the subtype should be equal to the subtype's Size. This applies to stand-alone objects and to components (unless a component_clause or a Component_Size clause applies).

48.f/3 {*AI05-0229-1*} A component_clause or a Component_Size clause can cause an object to be smaller than its subtype's specified size. The aspect Pack cannot; if a component subtype's size is specified, this limits how tightly the composite object can be packed.

48.g The Size of a class-wide (tagged) subtype is unspecified, because it's not clear what it should mean; it should certainly not depend on all of the descendants that happen to exist in a given program. Note that this cannot be detected at compile time, because in a generic unit, it is not necessarily known whether a given subtype is class-wide. It might raise an exception on some implementations.

48.h **Ramification:** A Size clause for a numeric subtype need not affect the underlying numeric type. For example, if I say:

48.i
```
type S is range 1..2;
for S'Size use 64;
```

48.j I am not guaranteed that S'Base'Last >= 2**63–1, nor that intermediate results will be represented in 64 bits.

48.k **Reason:** There is no need to complicate implementations for this sort of thing, because the right way to affect the base range of a type is to use the normal way of declaring the base range:

48.l
```
type Big is range -2**63 .. 2**63 - 1;
subtype Small is Big range 1..1000;
```

48.m **Ramification:** The Size of a large unconstrained subtype (e.g. String'Size) is likely to raise Constraint_Error, since it is a nonstatic expression of type *universal_integer* that might overflow the largest signed integer type. There is no requirement that the largest integer type be able to represent the size in bits of the largest possible object.

48.n/3 **Aspect Description for Size (subtype):** Size in bits of a subtype.

Implementation Requirements

49 In an implementation, Boolean'Size shall be 1.

Implementation Advice

50/5 {*AI95-00051-02*} {*AI12-0059-1*} If the Size of a subtype is nonconfirming and allows for efficient independent addressability (see 9.10) on the target architecture, then the Object_Size of the subtype should have the same value in the absence of an explicit specification of a different value.

Paragraphs 51 and 52 were moved to the Implementation Advice for attribute Object_Size.

52.a.1/5 **Implementation Advice:** If the Size of a subtype is nonconfirming and allows for efficient independent addressability, then the Object_Size of the subtype (unless otherwise specified) should equal the Size of the subtype.

52.a **Ramification:** Thus, on a typical 32-bit machine, "**for** S'Size **use** 32;" will guarantee that aliased objects of subtype S, and components whose subtype is S, will have Size = 32 (assuming the implementation chooses to obey this Implementation Advice). On the other hand, if one writes, "**for** S2'Size **use** 5;" then stand-alone objects of subtype S2 will typically have their Size rounded up to ensure independent addressability.

52.b Note that "**for** S'Size **use** 32;" does not cause things like formal parameters to have Size = 32 — the implementation is allowed to make all parameters be at least 64 bits, for example.

Note that "**for** S2'Size **use** 5;" requires record components whose subtype is S2 to be exactly 5 bits if the record type is packed. The same is not true of array components; their Size may be rounded up to the nearest factor of the word size. 52.c

Implementation Note: {*AI95-00291-02*} On most machines, arrays don't contain gaps between elementary components; if the Component_Size is greater than the Size of the component subtype, the extra bits are generally considered part of each component, rather than gaps between components. On the other hand, a record might contain gaps between elementary components, depending on what sorts of loads, stores, and masking operations are generally done by the generated code. 52.d/2

{*AI95-00291-02*} For an array, any extra bits stored for each elementary component will generally be part of the component — the whole point of storing extra bits is to make loads and stores more efficient by avoiding the need to mask out extra bits. The PDP-10 is one counter-example; since the hardware supports byte strings with a gap at the end of each word, one would want to pack in that manner. 52.e/2

Reason: {*AI12-0059-1*} We talk about the explicit specification of Object_Size so that we have specified what happens if both Object_Size and Size are specified incompatibly; we give priority to Object_Size. Note that the value of Size no longer has any direct effect on the Size of objects; what happens instead is that the value of Size can have an effect on the value of Object_Size in the absence of a specification for Object_Size, and it is always the value of Object_Size that determines the size of an object. 52.f/5

A Size clause on a composite subtype should not affect the internal layout of components. 53

Implementation Advice: A Size clause on a composite subtype should not affect the internal layout of components. 53.a.1/2

Reason: {*AI05-0229-1*} That's what Pack aspects, record_representation_clauses, and Component_Size clauses are for. 53.a/3

The recommended level of support for the Size attribute of subtypes is: 54

- {*AI12-0445-1*} The Size (if not specified) of a static discrete or fixed point subtype should be the number of bits necessary to represent each value belonging to the subtype using an unbiased representation, leaving space for a sign bit only if the subtype contains negative values. If such a subtype is a first subtype, then an implementation should support a specified Size for it that reflects this representation. 55/5

  Implementation Note: This applies to static enumeration subtypes, using the internal codes used to represent the values. 55.a

  For a two's-complement machine, this implies that for a static signed integer subtype S, if all values of S are in the range 0 .. 2^n–1, or all values of S are in the range $–2^{n–1}$ .. $2^{n–1}$–1, for some *n* less than or equal to the word size, then S'Size should be <= the smallest such *n*. For a one's-complement machine, it is the same except that in the second range, the lower bound "$–2^{n–1}$" is replaced by "$–2^{n–1}$+1". 55.b

  If an integer subtype (whether signed or unsigned) contains no negative values, the Size should not include space for a sign bit. 55.c

  Typically, the implementation will choose to make the Size of a subtype be exactly the smallest such *n*. However, it might, for example, choose a biased representation, in which case it could choose a smaller value. 55.d

  {*AI05-0229-1*} On most machines, it is in general not a good idea to pack (parts of) multiple stand-alone objects into the same storage element, because (1) it usually doesn't save much space, and (2) it requires locking to prevent tasks from interfering with each other, since separate stand-alone objects are independently addressable. Therefore, if S'Size = 2 on a machine with 8-bit storage elements, the size of a stand-alone object of subtype S will probably not be 2. It might, for example, be 8, 16 or 32, depending on the availability and efficiency of various machine instructions. The same applies to components of composite types, unless Pack, Component_Size, or record layout is specified. 55.e/3

  For an unconstrained discriminated object, if the implementation allocates the maximum possible size, then the Size attribute should return that maximum possible size. 55.f

  Ramification: The Size of an object X is not usually the same as that of its subtype S. If X is a stand-alone object or a parameter, for example, most implementations will round X'Size up to a storage element boundary, or more, so X'Size might be greater than S'Size. On the other hand, X'Size cannot be less than S'Size, even if the implementation can prove, for example, that the range of values actually taken on by X during execution is smaller than the range of S. 55.g

  For example, if S is a first integer subtype whose range is 0..3, S'Size will be probably be 2 bits, and components of packed composite types of this subtype will be 2 bits (assuming Storage_Unit is a multiple of 2), but stand-alone objects and parameters will probably not have a size of 2 bits; they might be rounded up to 32 bits, for example. On the other hand, Unchecked_Conversion will use the 2-bit size, even when converting a stand-alone object, as one would expect. 55.h

  Another reason for making the Size of an object bigger than its subtype's Size is to support the run-time detection of uninitialized variables. The implementation might add an extra value to a discrete subtype that represents the 55.i

uninitialized state, and check for this value on use. In some cases, the extra value will require an extra bit in the representation of the object. Such detection is not required by the language. If it is provided, the implementation has to be able to turn it off. For example, if the programmer gives a record_representation_clause or Component_Size clause that makes a component too small to allow the extra bit, then the implementation will not be able to perform the checking (not using this method, anyway).

55.j The fact that the size of an object is not necessarily the same as its subtype can be confusing:

55.k
```
type Device_Register is range 0..2**8 - 1;
for Device_Register'Size use 8; -- Confusing!
My_Device : Device_Register;
for My_Device'Address use To_Address(16#FF00#);
```

55.l/5 {*AI12-0005-1*} The programmer might think that My_Device'Size is 8, and that My_Device'Address points at an 8-bit location. However, this is not guaranteed to be true. In Ada 83 (and in Ada 95), My_Device'Size might well be 32, and My_Device'Address might well point at the high-order 8 bits of the 32-bit object, which are always all zero bits. If My_Device'Address is passed to an assembly language subprogram, based on the programmer's assumption, the program will not work properly.

55.m **Reason:** It is not reasonable to require that an implementation allocate exactly 8 bits to all objects of subtype Device_Register. For example, in many run-time models, stand-alone objects and parameters are always aligned to a word boundary. Such run-time models are generally based on hardware considerations that are beyond the control of the implementer. (It is reasonable to require that an implementation allocate exactly 8 bits to all components of subtype Device_Register, if packed.)

55.n **Ramification:** The correct way to write the above code is like this:

55.o
```
type Device_Register is range 0..2**8 - 1;
My_Device : Device_Register;
for My_Device'Size use 8;
for My_Device'Address use To_Address(16#FF00#);
```

55.p If the implementation cannot accept 8-bit stand-alone objects, then this will be illegal. However, on a machine where an 8-bit device register exists, the implementation will probably be able to accept 8-bit stand-alone objects. Therefore, My_Device'Size will be 8, and My_Device'Address will point at those 8 bits, as desired.

55.q If an object of subtype Device_Register is passed to a foreign language subprogram, it will be passed according to that subprogram's conventions. Most foreign language implementations have similar run-time model restrictions. For example, when passing to a C function, where the argument is of the C type char* (that is, pointer to char), the C compiler will generally expect a full word value, either on the stack, or in a register. It will *not* expect a single byte. Thus, Size clauses for subtypes really have nothing to do with passing parameters to foreign language subprograms.

56 - For a subtype implemented with levels of indirection, the Size should include the size of the pointers, but not the size of what they point at.

56.a **Ramification:** For example, if a task object is represented as a pointer to some information (including a task stack), then the size of the object should be the size of the pointer. The Storage_Size, on the other hand, should include the size of the stack.

56.1/2 - {*AI95-00051-02*} An implementation should support a Size clause for a discrete type, fixed point type, record type, or array type, subject to the following:

  56.2/5 - {*AI95-00051-02*} {*AI12-0444-1*} An implementation is not required to support a Size clause for a signed integer type specifying a Size greater than that of the largest signed integer type supported by the implementation in the absence of a size clause (that is, when the size is chosen by default). A corresponding limitation may be imposed for modular integer types, fixed point types, enumeration types, record types, and array types.

56.b/2 **Discussion:** {*AI95-00051-02*} Note that the "corresponding limitation" for a record or array type implies that an implementation may impose some reasonable maximum size for records and arrays (e.g. 2**32 bits), which is an upper bound ("capacity" limit) on the size, whether chosen by default or by being specified by the user. The largest size supported for records need not be the same as the largest size supported for arrays.

56.b.1/3 **Ramification:** {*AI05-0155-1*} Only Size clauses with a size greater than or equal to the Size that would be chosen by default may be safely presumed to be supported on nonstatic elementary subtypes. Implementations may choose to support smaller sizes, but only if the Size allows any value of the subtype to be represented, for any possible value of the bounds.

  56.3/5 - {*AI95-00291-02*} {*AI12-0444-1*} A nonconfirming size clause for the first subtype of a derived untagged by-reference type is not required to be supported.

Implementation Advice: The recommended level of support for the Size attribute should be followed. 56.c/2

Ramification: {*AI95-00291-02*} There is no recommendation to support any nonconfirming Size clauses for types not mentioned above. Remember that 13.1 requires support for confirming Size clauses for all types. 56.d/2

NOTE 5 Size is a subtype-specific attribute. 57

NOTE 6 {*AI05-0229-1*} A component_clause or Component_Size clause can override a specified Size. Aspect Pack cannot. 58/3

Inconsistencies With Ada 83

{*AI95-00114-01*} We specify the meaning of Size in much more detail than Ada 83. This is not technically an inconsistency, but it is in practice, as most Ada 83 compilers use a different definition for Size than is required here. This should have been documented more explicitly during the Ada 9X process. 58.a.1/2

Wording Changes from Ada 83

The requirement for a nonnegative value in a Size clause was not in RM83, but it's hard to see how it would make sense. For uniformity, we forbid negative sizes, rather than letting implementations define their meaning. 58.a

Static Semantics

{*AI12-0059-1*} For every subtype S: 58.1/5

S'Object_Size 58.2/5

{*AI12-0059-1*} If S is definite, denotes the size (in bits) of a stand-alone aliased object, or a component of subtype S in the absence of an aspect_specification or representation item that specifies the size of the object or component. If S is indefinite, the meaning is implementation-defined. The value of this attribute is of the type *universal_integer*. If not specified otherwise, the Object_Size of a subtype is at least as large as the Size of the subtype. Object_Size may be specified via an attribute_definition_clause; the expression of such a clause shall be static and its value nonnegative. All aliased objects with nominal subtype S have the size S'Object_Size. In the absence of an explicit specification, the Object_Size of a subtype S defined by a subtype_indication without a constraint, is that of the value of the Object_Size of the subtype denoted by the subtype_mark of the subtype_indication, at the point of this definition.

Implementation defined: The meaning of Object_Size for indefinite subtypes. 58.a.1/5

Ramification: We allow the specification of Object_Size for any subtype (*not* just first subtypes, as with other aspects). An implementation can, of course, put restrictions on which subtypes allow specification of Object_Size (as with any representation attribute). 58.b/5

Implementation Advice

{*AI12-0059-1*} If S is a definite first subtype and S'Object_Size is not specified, S'Object_Size should be the smallest multiple of the storage element size larger than or equal to S'Size that is consistent with the alignment of S. 58.3/5

Implementation Advice: If S is a definite first subtype for which Object_Size is not specified, S'Object_Size should be the smallest multiple of the storage element size larger than or equal to S'Size that is consistent with the alignment of S. 58.b.1/5

Reason: Many implementations “round up” allocations to the nearest multiple of the alignment. For instance, this example appears in the GNAT documentation: 58.c/5

[Given] a record containing an integer and a character: 58.d/5

```
type Rec is record                                                    58.e/5
   I : Integer;
   C : Character;
end record;
```

will have a size of 40 (that is Rec'Size will be 40). The alignment will be 4, because of the integer field, and so the default size of record objects for this type will be 64 (8 bytes). 58.f/5

We purposely used the vague phrase “consistent with the alignment of S” so that implementations that do not round up allocations (just using padding to provide alignment) are not required to do any rounding up. 58.g/5

58.4/5 {*AI12-0059-1*} If X denotes an object (including a component) of subtype S, X'Size should equal S'Object_Size, unless:

58.5/5 • X'Size is specified; or

58.6/5 • X is a nonaliased stand-alone object; or

58.7/5 • The size of X is determined by a component_clause or Component_Size clause; or

58.8/5 • The type containing component X is packed.

58.g.1/5 **Implementation Advice:** The Size of most objects of a subtype should equal the Object_Size of the subtype.

58.9/5 {*AI12-0059-1*} An Object_Size clause on a composite type should not affect the internal layout of components.

58.g.2/5 **Implementation Advice:** An Object_Size clause on a composite type should not affect the internal layout of components.

58.10/5 {*AI12-0059-1*} The recommended level of support for the Object_Size attribute of subtypes is:

58.11/5 • If S is a static signed integer subtype, the implementation should support the specification of S'Object_Size to match the size of any signed integer base subtype provided by the implementation that is no smaller than S'Size. Corresponding support is expected for modular integer subtypes, fixed point subtypes, and enumeration subtypes.

58.h/5 **Ramification:** The intent is that a compiler need support only those Object_Sizes for integer subtypes that it might select for an integer type declaration. Similarly for the other kinds of subtypes listed.

58.12/5 • If S is an array or record subtype with static constraints and S is not a first subtype of a derived untagged by-reference type, the implementation should support the specification of S'Object_Size to be any multiple of the storage element size that is consistent with the alignment of S, that is no smaller than S'Size, and that is no larger than that of the largest composite subtype supported by the implementation.

58.i/5 **Discussion:** Any extra bits required this way will be padding bits. Unlike elementary objects, padding bits can be considered part of composite objects.

58.13/5 • If S is some other subtype, only confirming specifications of Object_Size are required to be supported.

58.i.1/5 **Implementation Advice:** The recommended level of support for the Object_Size attribute should be followed.

Static Semantics

59/1 For a prefix T that denotes a task object [(after any implicit dereference)]:

60/3 T'Storage_Size

{*AI05-0229-1*} Denotes the number of storage elements reserved for the task. The value of this attribute is of the type *universal_integer*. The Storage_Size includes the size of the task's stack, if any. The language does not specify whether or not it includes other storage associated with the task (such as the “task control block” used by some implementations.) If the aspect Storage_Size is specified for the type of the object, the value of the Storage_Size attribute is at least the value determined by the aspect.

60.a **Ramification:** The value of this attribute is never negative, since it is impossible to “reserve” a negative number of storage elements.

60.b If the implementation chooses to allocate an initial amount of storage, and then increase this as needed, the Storage_Size cannot include the additional amounts (assuming the allocation of the additional amounts can raise Storage_Error); this is inherent in the meaning of “reserved”.

60.c The implementation is allowed to allocate different amounts of storage for different tasks of the same subtype.

60.d Storage_Size is also defined for access subtypes — see 13.11.

61/3 {*AI95-0229-1*} [Aspect Storage_Size specifies the amount of storage to be reserved for the execution of a task.]

Paragraphs 62 through 65 were moved to Annex J, "Obsolescent Features".

Static Semantics

{*AI05-0229-1*} {*AI05-0269-1*} For a task type (including the anonymous type of a single_task_declaration), the following language-defined representation aspect may be specified: 65.1/3

Storage_Size 65.2/3

The Storage_Size aspect is an expression, which shall be of any integer type.

To be honest: This definition somewhat conflicts with the "automatic" one for the obsolescent attribute Storage_Size (which can be specified). The only difference is where the given expression is evaluated. We intend for the above definition to supersede that "automatic" definition for this attribute. 65.a/3

Ramification: Note that the value of the Storage_Size aspect *is* an expression; it is not the *value* of an expression. The expression is evaluated for each object of the type (see below). 65.b/3

Aspect Description for Storage_Size (task): Size in storage elements reserved for a task type or single task object. 65.c/3

Legality Rules

{*AI05-0229-1*} The Storage_Size aspect shall not be specified for a task interface type. 65.3/3

Dynamic Semantics

{*AI05-0229-1*} When a task object is created, the expression (if any) associated with the Storage_Size aspect of its type is evaluated; the Storage_Size attribute of the newly created task object is at least the value of the expression. 66/3

Ramification: The implementation is allowed to round up a specified Storage_Size amount. For example, if the implementation always allocates in chunks of 4096 bytes, the number 200 might be rounded up to 4096. Also, if the user specifies a negative number, the implementation has to normalize this to 0, or perhaps to a positive number. 66.a

{*AI05-0229-1*} If the Storage_Size aspect is not specified for the type of the task object, the value of the Storage_Size attribute is unspecified. 66.b/3

At the point of task object creation, or upon task activation, Storage_Error is raised if there is insufficient free storage to accommodate the requested Storage_Size. 67

Static Semantics

For a prefix X that denotes an array subtype or array object [(after any implicit dereference)]: 68/1

X'Component_Size 69

Denotes the size in bits of components of the type of X. The value of this attribute is of type *universal_integer*.

Component_Size may be specified for array types via an attribute_definition_clause; the expression of such a clause shall be static, and its value nonnegative. 70

Implementation Note: The intent is that the value of X'Component_Size is always nonnegative. If the array is stored "backwards" in memory (which might be caused by an implementation-defined pragma), X'Component_Size is still positive. 70.a

Ramification: For an array object A, A'Component_Size = A(I)'Size for any index I. 70.b

Aspect Description for Component_Size: Size in bits of a component of an array type. 70.c/3

Implementation Advice

The recommended level of support for the Component_Size attribute is: 71

- {*AI12-0444-1*} An implementation is not required to support specified Component_Sizes that are less than the Size of the component subtype. 72/5
- {*AI05-0229-1*} An implementation should support specified Component_Sizes that are factors and multiples of the word size. For such Component_Sizes, the array should contain no gaps 73/3

between components. For other Component_Sizes (if supported), the array should contain no gaps between components when Pack is also specified; the implementation should forbid this combination in cases where it cannot support a no-gaps representation.

73.a/3 **Ramification:** {*AI05-0229-1*} For example, if Storage_Unit = 8, and Word_Size = 32, then the user is allowed to specify a Component_Size of 1, 2, 4, 8, 16, and 32, with no gaps. In addition, *n**32 is allowed for positive integers *n*, again with no gaps. If the implementation accepts Component_Size = 3, then it might allocate 10 components per word, with a 2-bit gap at the end of each word (unless Pack is also specified), or it might not have any internal gaps at all. (There can be gaps at either end of the array.)

73.b/2 **Implementation Advice:** The recommended level of support for the Component_Size attribute should be followed.

Static Semantics

73.1/3 {*AI05-0191-1*} For a prefix X that denotes an object:

73.2/4 X'Has_Same_Storage

{*AI05-0191-1*} X'Has_Same_Storage denotes a function with the following specification:

73.3/3
```
function X'Has_Same_Storage (Arg : any_type)
  return Boolean
```

73.4/4 {*AI05-0191-1*} {*AI05-0264-1*} {*AI12-0077-1*} The actual parameter shall be a name that denotes an object. The object denoted by the actual parameter can be of any type. This function evaluates the names of the objects involved. It returns True if the representation of the object denoted by the actual parameter occupies exactly the same bits as the representation of the object denoted by X and the objects occupy at least one bit; otherwise, it returns False.

73.c/3 **Discussion:** Has_Same_Storage means that, if the representation is contiguous, the objects sit at the same address and occupy the same length of memory.

73.5/3 {*AI05-0191-1*} For a prefix X that denotes an object:

73.6/3 X'Overlaps_Storage

{*AI05-0191-1*} X'Overlaps_Storage denotes a function with the following specification:

73.7/3
```
function X'Overlaps_Storage (Arg : any_type)
  return Boolean
```

73.8/3 {*AI05-0191-1*} {*AI05-0264-1*} The actual parameter shall be a name that denotes an object. The object denoted by the actual parameter can be of any type. This function evaluates the names of the objects involved and returns True if the representation of the object denoted by the actual parameter shares at least one bit with the representation of the object denoted by X; otherwise, it returns False.

73.9/3 NOTE 7 {*AI05-0191-1*} X'Has_Same_Storage(Y) implies X'Overlaps_Storage(Y).

73.10/3 NOTE 8 {*AI05-0191-1*} X'Has_Same_Storage(Y) and X'Overlaps_Storage(Y) are not considered to be reads of X and Y.

Static Semantics

73.11/3 {*8652/0009*} {*AI95-00137-01*} {*AI05-0183-1*} The following type-related operational attribute is defined: External_Tag.

74/1 {*8652/0009*} {*AI95-00137-01*} For every subtype S of a tagged type *T* (specific or class-wide):

75/3 S'External_Tag

{*8652/0040*} {*AI95-00108-01*} {*AI05-0092-1*} S'External_Tag denotes an external string representation for S'Tag; it is of the predefined type String. External_Tag may be specified for a specific tagged type via an attribute_definition_clause; the expression of such a clause shall be static. The default external tag representation is implementation defined. See 13.13.2. The value of External_Tag is never inherited[; the default value is always used unless a new value is directly specified for a type].

75.a **Implementation defined:** The default external representation for a type tag.

Aspect Description for External_Tag: Unique identifier for a tagged type in streams. 75.b/3

Dynamic Semantics

{*AI05-0113-1*} If a user-specified external tag S'External_Tag is the same as T'External_Tag for some other tagged type declared by a different declaration in the partition, Program_Error is raised by the elaboration of the attribute_definition_clause. 75.1/3

Ramification: This rule does not depend on the visibility of the other tagged type, but it does depend on the existence of the other tagged type. The other tagged type could have the default external tag or a user-specified external tag. 75.c/3

This rule allows the same declaration to be elaborated multiple times. In that case, different types could have the same external tag. If that happens, Internal_Tag would return some unspecified tag, and Descendant_Tag probably would return the intended tag (using the given ancestor to determine which type is intended). However, in some cases (such as multiple instantiations of a derived tagged type declared in a generic body), Tag_Error might be raised by Descendant_Tag if multiple types are identified. 75.d/3

Note that while there is a race condition inherent in this definition (which attribute_definition_clause raises Program_Error depends on the order of elaboration), it doesn't matter as a program with two such clauses is simply wrong. Two types that both come from the same declaration are allowed, as noted previously. 75.e/3

Implementation Requirements

In an implementation, the default external tag for each specific tagged type declared in a partition shall be distinct, so long as the type is declared outside an instance of a generic body. If the compilation unit in which a given tagged type is declared, and all compilation units on which it semantically depends, are the same in two different partitions, then the external tag for the type shall be the same in the two partitions. What it means for a compilation unit to be the same in two different partitions is implementation defined. At a minimum, if the compilation unit is not recompiled between building the two different partitions that include it, the compilation unit is considered the same in the two partitions. 76

Implementation defined: What determines whether a compilation unit is the same in two different partitions. 76.a

Reason: These requirements are important because external tags are used for input/output of class-wide types. These requirements ensure that what is written by one program can be read back by some other program so long as they share the same declaration for the type (and everything it depends on). 76.b

The user may specify the external tag if (s)he wishes its value to be stable even across changes to the compilation unit in which the type is declared (or changes in some unit on which it depends). 76.c

{*AI95-00114-01*} We use a String rather than a Stream_Element_Array to represent an external tag for portability. 76.d/2

Ramification: Note that the characters of an external tag need not all be graphic characters. In other words, the external tag can be a sequence of arbitrary 8-bit bytes. 76.e

Implementation Permissions

{*AI05-0113-1*} If a user-specified external tag S'External_Tag is the same as T'External_Tag for some other tagged type declared by a different declaration in the partition, the partition may be rejected. 76.1/3

Ramification: This is, in general, a post-compilation check. This permission is intended for implementations that do link-time construction of the external tag lookup table; implementations that dynamically construct the table will likely prefer to raise Program_Error upon elaboration of the problem construct. We don't want this check to require any implementation complexity, as it will be very rare that there would be a problem. 76.f/3

NOTE 9 {*AI95-00270-01*} The following language-defined attributes are specifiable, at least for some of the kinds of entities to which they apply: Address, Alignment, Bit_Order, Component_Size, External_Tag, Input, Machine_Radix, Output, Read, Size, Small, Storage_Pool, Storage_Size, Stream_Size, and Write. 77/2

NOTE 10 It follows from the general rules in 13.1 that if one writes "**for** X'Size **use** Y;" then the X'Size attribute_reference will return Y (assuming the implementation allows the Size clause). The same is true for all of the specifiable attributes except Storage_Size. 78

Ramification: An implementation may specify that an implementation-defined attribute is specifiable for certain entities. This follows from the fact that the semantics of implementation-defined attributes is implementation defined. An implementation is not allowed to make a language-defined attribute specifiable if it isn't. 78.a

Examples

79 *Examples of attribute definition clauses:*

80
```
Byte : constant := 8;
Page : constant := 2**12;
```

81
```
type Medium is range 0 .. 65_000;
for Medium'Size use 2*Byte;
for Medium'Alignment use 2;
Device_Register : Medium;
for Device_Register'Size use Medium'Size;
for Device_Register'Address use
   System.Storage_Elements.To_Address(16#FFFF_0020#);
```

82
```
type Short is delta 0.01 range -100.0 .. 100.0;
for Short'Size use 15;
```

83
```
for Car_Name'Storage_Size use -- specify access type's storage pool size
        2000*((Car'Size/System.Storage_Unit) +1); -- approximately 2000 cars
```

84/2
```
{AI95-00441-01} function My_Input(Stream : not null access
Ada.Streams.Root_Stream_Type'Class)
   return T;
for T'Input use My_Input; -- see 13.13.2
```

85/5 {*AI12-0452-1*} In the Size clause for Short, fifteen bits is the minimum necessary, since the type definition requires Short'Small <= 2**(–7).

Extensions to Ada 83

86.a The syntax rule for length_clause is replaced with the new syntax rule for attribute_definition_clause, and it is modified to allow a name (as well as an expression).

Wording Changes from Ada 83

86.b The syntax rule for attribute_definition_clause now requires that the prefix of the attribute be a local_name; in Ada 83 this rule was stated in the text.

86.c/2 {*AI95-00114-01*} In Ada 83, the relationship between a aspect_clause specifying a certain aspect and an attribute that queried that aspect was unclear. In Ada 95, they are the same, except for certain explicit exceptions.

Wording Changes from Ada 95

86.d/2 {*8652/0009*} {*AI95-00137-01*} **Corrigendum:** Added wording to specify for each attribute whether it is an operational or representation attribute.

86.e/2 {*8652/0040*} {*AI95-00108-01*} **Corrigendum:** Added wording to specify that External_Tag is never inherited.

86.f/2 {*AI95-00051-01*} {*AI95-00291-01*} Adjusted the Recommended Level of Support for Alignment to eliminate nonsense requirements and to ensure that useful capabilities are required.

86.g/2 {*AI95-00051-01*} {*AI95-00291-01*} Adjusted the Recommended Level of Support for Size to eliminate nonsense requirements and to ensure that useful capabilities are required. Also eliminated any dependence on whether an aspect was specified (a confirming representation item should not affect the semantics).

86.h/2 {*AI95-00133-01*} Added the definition of machine scalar.

86.i/2 {*AI95-00247-01*} Removed the requirement that specified alignments for a composite type cannot override those for their components, because it was never intended to apply to components whose location was specified with a representation item. Moreover, it causes a difference in legality when a confirming alignment is specified for one of the composite types.

86.j/2 {*AI95-00291-02*} Removed recommended level of support rules about types with by-reference and aliased parts, because there are now blanket rules covering all recommended level of support rules.

86.k/2 {*AI95-00291-02*} Split the definition of Alignment for subtypes and for objects. This simplified the wording and eliminated confusion about which rules applied to objects, which applied to subtypes, and which applied to both.

Inconsistencies With Ada 2005

86.l/3 {*AI95-0095-1*} **Correction:** An address attribute with a prefix of a generic formal subprogram whose actual parameter has convention Intrinsic now raises Program_Error. Since it is unlikely that such an attribute would have done anything

useful (a subprogram with convention Intrinsic is not expected to have a normal subprogram body), it is highly unlikely that any existing programs would notice the difference, and any that do probably are buggy.

{*AI95-0113-1*} **Correction:** User-specified external tags that conflict with other external tags raise Program_Error (or are optionally illegal). This was legal and did not raise an exception in the past, although the effects were not defined. So while a program might depend on such behavior, the results were not portable (even to different versions of the same implementation). Such programs should be rare. 86.m/3

Incompatibilities With Ada 2005

{*AI05-0095-1*} **Correction:** An address attribute with a prefix of a subprogram with convention Intrinsic is now illegal. Such attributes are very unlikely to have provided a useful answer (the intended meaning of convention Intrinsic is that there is no actual subprogram body for the operation), so this is highly unlikely to affect any existing programs unless they have a hidden bug. 86.n/3

Extensions to Ada 2005

{*AI05-0191-1*} Attributes Has_Same_Storage and Overlaps_Storage are new. 86.o/3

{*AI05-0229-1*} Aspect Storage_Size is new; pragma Storage_Size is now obsolescent, joining attribute Storage_Size for task types. 86.p/3

Wording Changes from Ada 2005

{*AI05-0009-1*} **Correction:** Improved the description of erroneous execution for address clauses to make it clear that specifying an address inappropriate for the entity will lead to erroneous execution. 86.q/3

{*AI05-0116-1*} **Correction:** Added Implementation Advice for the alignment of class-wide types. 86.r/3

Extensions to Ada 2012

{*AI12-0059-1*} Attribute Object_Size is new. 86.s/5

Wording Changes from Ada 2012

{*AI12-0070-1*} **Corrigendum:** Clarified the behavior of Has_Same_Storage when 'Size = 0. 86.t/4

13.4 Enumeration Representation Clauses

[An enumeration_representation_clause specifies the internal codes for enumeration literals.] 1

Syntax

enumeration_representation_clause ::= 2
  for *first_subtype*_local_name **use** enumeration_aggregate;

enumeration_aggregate ::= array_aggregate 3

Name Resolution Rules

The enumeration_aggregate shall be written as a one-dimensional array_aggregate, for which the index subtype is the unconstrained subtype of the enumeration type, and each component expression is expected to be of any integer type. 4

Ramification: The "full coverage rules" for aggregates applies. An **others** is not allowed — there is no applicable index constraint in this context. 4.a

Legality Rules

The *first_subtype*_local_name of an enumeration_representation_clause shall denote an enumeration subtype. 5

Ramification: As for all type-related representation items, the local_name is required to denote a first subtype. 5.a

{*AI95-00287-01*} Each component of the array_aggregate shall be given by an expression rather than a <>. The expressions given in the array_aggregate shall be static, and shall specify distinct integer codes 6/2

for each value of the enumeration type; the associated integer codes shall satisfy the predefined ordering relation of the type.

6.a **Reason:** Each value of the enumeration type has to be given an internal code, even if the first subtype of the enumeration type is constrained to only a subrange (this is only possible if the enumeration type is a derived type). This "full coverage" requirement is important because one may refer to Enum'Base'First and Enum'Base'Last, which need to have defined representations.

Static Semantics

7 An enumeration_representation_clause specifies the *coding* aspect of representation. The coding consists of the *internal code* for each enumeration literal, that is, the integral value used internally to represent each literal.

7.a/3 **Aspect Description for Coding:** Internal representation of enumeration literals. Specified by an enumeration_representation_clause, not by an aspect_specification.

Implementation Requirements

8 For nonboolean enumeration types, if the coding is not specified for the type, then for each value of the type, the internal code shall be equal to its position number.

8.a **Reason:** This default representation is already used by all known Ada compilers for nonboolean enumeration types. Therefore, we make it a requirement so users can depend on it, rather than feeling obliged to supply for every enumeration type an enumeration representation clause that is equivalent to this default rule.

8.b **Discussion:** For boolean types, it is relatively common to use all ones for True, and all zeros for False, since some hardware supports that directly. Of course, for a one-bit Boolean object (like in a packed array), False is presumably zero and True is presumably one (choosing the reverse would be extremely unfriendly!).

Implementation Advice

9 The recommended level of support for enumeration_representation_clauses is:

10/5 - {*AI12-0444-1*} An implementation should support at least the internal codes in the range System.Min_Int .. System.Max_Int. An implementation is not required to support enumeration_representation_clauses for boolean types.

10.a **Ramification:** The implementation may support numbers outside the above range, such as numbers greater than System.Max_Int. See AI83-00564.

10.b **Reason:** The benefits of specifying the internal coding of a boolean type do not outweigh the implementation costs. Consider, for example, the implementation of the logical operators on a packed array of booleans with strange internal codes. It's implementable, but not worth it.

10.c/2 **Implementation Advice:** The recommended level of support for enumeration_representation_clauses should be followed.

Static Semantics

10.1/5 {*AI12-0237-1*} For every discrete subtype S, the following attributes are defined:

10.2/5 S'Enum_Rep

{*AI12-0237-1*} S'Enum_Rep denotes a function with the following specification:

10.3/5
```
function S'Enum_Rep (Arg : S'Base) return universal_integer
```

10.4/5 This function returns the representation value of the value of Arg, as a value of type *universal_integer*. The *representation value* is the internal code specified in an enumeration representation clause, if any, for the type corresponding to the value of Arg, and otherwise is the position number of the value.

10.5/5 S'Enum_Val

{*AI12-0237-1*} S'Enum_Val denotes a function with the following specification:

10.6/5
```
function S'Enum_Val (Arg : universal_integer) return S'Base
```

This function returns a value of the type of S whose representation value equals the value of Arg. For the evaluation of a call on S'Enum_Val, if there is no value in the base range of its type with the given representation value, Constraint_Error is raised. 10.7/5

Reason: We define these on all discrete types so that they can be used inside of a generic unit on a subtype of a generic formal discrete type. They're not useful on integer types (they have the same effect as S'Pos and S'Val). 10.d/5

NOTE {*8652/0009*} {*AI95-00137-01*} {*AI05-0299-1*} {*AI12-0237-1*} {*AI12-0442-1*} Attribute Enum_Rep can be used to query the internal codes used for an enumeration type; attribute Enum_Val can be used to convert from an internal code to an enumeration value. The other attributes of the type, such as Succ, Pred, and Pos, are unaffected by an enumeration_representation_clause. For example, Pos always returns the position number, *not* an internal integer code that was specified in an enumeration_representation_clause. 11/5

Discussion: Suppose the enumeration type in question is derived: 11.a

```
type T1 is (Red, Green, Blue);
subtype S1 is T1 range Red .. Green;
type S2 is new S1;
for S2 use (Red => 10, Green => 20, Blue => 30);
```
11.b

{*8652/0009*} {*AI95-00137-01*} The enumeration_representation_clause has to specify values for all enumerals, even ones that are not in S2 (such as Blue). The Base attribute can be used to get at these values. For example: 11.c/1

```
for I in S2'Base loop
    ... -- When I equals Blue, the internal code is 30.
end loop;
```
11.d

We considered allowing or requiring "**for** S2'Base **use** ..." in cases like this, but it didn't seem worth the trouble. 11.e

Examples

{*AI12-0312-1*} *Examples of enumeration representation clauses:* 12/5

```
type Mix_Code is (ADD, SUB, MUL, LDA, STA, STZ);
```
13

```
for Mix_Code use
   (ADD => 1, SUB => 2, MUL => 3, LDA => 8, STA => 24, STZ =>33);
```
14

{*AI12-0312-1*} -- *See 3.5.2.*

```
for Roman_Digit use ('I' => 1,
                     'V' => 5,
                     'X' => 10,
                     'L' => 50,
                     'C' => 100,
                     'D' => 500,
                     'M' => 1000);
```
15/5

{*AI12-0312-1*} -- *For an example of the use of attribute Enum_Rep, see 4.2.1.* 16/5

Extensions to Ada 83

As in other similar contexts, Ada 95 allows expressions of any integer type, not just expressions of type *universal_integer*, for the component expressions in the enumeration_aggregate. The preference rules for the predefined operators of *root_integer* eliminate any ambiguity. 16.a

For portability, we now require that the default coding for an enumeration type be the "obvious" coding using position numbers. This is satisfied by all known implementations. 16.b

Wording Changes from Ada 95

{*8652/0009*} {*AI95-00137-01*} **Corrigendum:** Updated to reflect that we no longer have something called representation_clause. 16.c/2

{*AI95-00287-01*} Added wording to prevent the use of <> in a enumeration_representation_clause. (<> is newly added to array_aggregates.) 16.d/2

Extensions to Ada 2012

{*AI12-0237-1*} Attributes Enum_Rep and Enum_Val are new. 16.e/5

13.5 Record Layout

1 The *(record) layout* aspect of representation consists of the *storage places* for some or all components, that is, storage place attributes of the components. The layout can be specified with a record_representation_clause.

13.5.1 Record Representation Clauses

1 [A record_representation_clause specifies the storage representation of records and record extensions, that is, the order, position, and size of components (including discriminants, if any).]

Language Design Principles

1.a/2 {*AI95-00114-01*} It should be feasible for an implementation to use negative offsets in the representation of composite types. However, no implementation should be forced to support negative offsets. Therefore, in the interest of uniformity, negative offsets should be disallowed in record_representation_clauses.

Syntax

2/5 {*AI12-0213-1*} record_representation_clause ::=
for *first_subtype*_local_name **use**
record [mod_clause]
{component_clause}
end record [local_name];

3 component_clause ::=
*component*_local_name **at** position **range** first_bit .. last_bit;

4 position ::= *static*_expression

5 first_bit ::= *static*_simple_expression

6 last_bit ::= *static*_simple_expression

6.a **Reason:** First_bit and last_bit need to be simple_expression instead of expression for the same reason as in range (see 3.5, “Scalar Types”).

6.1/5 {*AI12-0213-1*} If a local_name appears at the end of the record_representation_clause, it shall repeat the *first_subtype*_local_name.

Name Resolution Rules

7 Each position, first_bit, and last_bit is expected to be of any integer type.

7.a **Ramification:** These need not have the same integer type.

Legality Rules

8/2 {*AI95-00436-01*} The *first_subtype*_local_name of a record_representation_clause shall denote a specific record or record extension subtype.

8.a **Ramification:** As for all type-related representation items, the local_name is required to denote a first subtype.

9 If the *component*_local_name is a direct_name, the local_name shall denote a component of the type. For a record extension, the component shall not be inherited, and shall not be a discriminant that corresponds to a discriminant of the parent type. If the *component*_local_name has an attribute_designator, the direct_name of the local_name shall denote either the declaration of the type or a component of the type, and the attribute_designator shall denote an implementation-defined implicit component of the type.

The position, first_bit, and last_bit shall be static expressions. The value of position and first_bit shall be nonnegative. The value of last_bit shall be no less than first_bit – 1. 10

Ramification: A component_clause such as "X **at** 4 **range** 0..–1;" is allowed if X can fit in zero bits. 10.a

{*AI95-00133-01*} If the nondefault bit ordering applies to the type, then either: 10.1/2

- the value of last_bit shall be less than the size of the largest machine scalar; or 10.2/2
- the value of first_bit shall be zero and the value of last_bit + 1 shall be a multiple of System.Storage_Unit. 10.3/2

At most one component_clause is allowed for each component of the type, including for each discriminant (component_clauses may be given for some, all, or none of the components). Storage places within a component_list shall not overlap, unless they are for components in distinct variants of the same variant_part. 11

A name that denotes a component of a type is not allowed within a record_representation_clause for the type, except as the *component_*local_name of a component_clause. 12

Reason: It might seem strange to make the record_representation_clause part of the declarative region, and then disallow mentions of the components within almost all of the record_representation_clause. The alternative would be to treat the *component_*local_name like a formal parameter name in a subprogram call (in terms of visibility). However, this rule would imply slightly different semantics, because (given the actual rule) the components can hide other declarations. This was the rule in Ada 83, and we see no reason to change it. The following, for example, was and is illegal: 12.a

```
type T is
    record
        X : Integer;
    end record;
X : constant := 31; -- Same defining name as the component.
for T use
    record
        X at 0 range 0..X; -- Illegal!
    end record;
```

12.b

The component X hides the named number X throughout the record_representation_clause. 12.c

Static Semantics

{*AI95-00133-01*} A record_representation_clause (without the mod_clause) specifies the layout. 13/2

Aspect Description for Layout (record): Layout of record components. Specified by a record_representation_clause, not by an aspect_specification. 13.a/3

Aspect Description for Record layout: See Layout. 13.b/3

{*AI95-00133-01*} If the default bit ordering applies to the type, the position, first_bit, and last_bit of each component_clause directly specify the position and size of the corresponding component. 13.1/2

{*AI95-00133-01*} {*AI05-0264-1*} If the nondefault bit ordering applies to the type, then the layout is determined as follows: 13.2/3

- the component_clauses for which the value of last_bit is greater than or equal to the size of the largest machine scalar directly specify the position and size of the corresponding component; 13.3/2
- for other component_clauses, all of the components having the same value of position are considered to be part of a single machine scalar, located at that position; this machine scalar has a size which is the smallest machine scalar size larger than the largest last_bit for all component_clauses at that position; the first_bit and last_bit of each component_clause are then interpreted as bit offsets in this machine scalar. 13.4/2

This paragraph was deleted.{*AI95-00133-01*} 13.c/2

Ramification: A component_clause also determines the value of the Size attribute of the component, since this attribute is related to First_Bit and Last_Bit. 13.d

14 [A record_representation_clause for a record extension does not override the layout of the parent part;] if the layout was specified for the parent type, it is inherited by the record extension.

Implementation Permissions

15 An implementation may generate implementation-defined components (for example, one containing the offset of another component). An implementation may generate names that denote such implementation-defined components; such names shall be implementation-defined attribute_references. An implementation may allow such implementation-defined names to be used in record_representation_clauses. An implementation can restrict such component_clauses in any manner it sees fit.

15.a **Implementation defined:** Implementation-defined components.

15.b **Ramification:** Of course, since the semantics of implementation-defined attributes is implementation defined, the implementation need not support these names in all situations. They might be purely for the purpose of component_clauses, for example. The visibility rules for such names are up to the implementation.

15.c We do not allow such component names to be normal identifiers — that would constitute blanket permission to do all kinds of evil things.

15.d **Discussion:** Such implementation-defined components are known in the vernacular as “dope”. Their main purpose is for storing offsets of components that depend on discriminants.

16 If a record_representation_clause is given for an untagged derived type, the storage place attributes for all of the components of the derived type may differ from those of the corresponding components of the parent type, even for components whose storage place is not specified explicitly in the record_representation_clause.

16.a **Reason:** This is clearly necessary, since the whole record may need to be laid out differently.

Implementation Advice

17 The recommended level of support for record_representation_clauses is:

17.1/2 • {*AI95-00133-01*} An implementation should support machine scalars that correspond to all of the integer, floating point, and address formats supported by the machine.

18 • An implementation should support storage places that can be extracted with a load, mask, shift sequence of machine code, and set with a load, shift, mask, store sequence, given the available machine instructions and run-time model.

19 • A storage place should be supported if its size is equal to the Size of the component subtype, and it starts and ends on a boundary that obeys the Alignment of the component subtype.

20/2 • {*AI95-00133-01*} For a component with a subtype whose Size is less than the word size, any storage place that does not cross an aligned word boundary should be supported.

20.a **Reason:** The above recommendations are sufficient to define interfaces to most interesting hardware. This causes less implementation burden than the definition in ACID, which requires arbitrary bit alignments of arbitrarily large components. Since the ACID definition is neither enforced by the ACVC, nor supported by all implementations, it seems OK for us to weaken it.

21 • An implementation may reserve a storage place for the tag field of a tagged type, and disallow other components from overlapping that place.

21.a **Ramification:** Similar permission for other dope is not granted.

22/5 • {*AI12-0444-1*} An implementation is not required to support a component_clause for a component of an extension part if the storage place is not after the storage places of all components of the parent type, whether or not those storage places had been specified.

22.a **Reason:** These restrictions are probably necessary if block equality operations are to be feasible for class-wide types. For block comparison to work, the implementation typically has to fill in any gaps with zero (or one) bits. If a “gap” in the parent type is filled in with a component in a type extension, then this won't work when a class-wide object is passed by reference, as is required.

Implementation Advice: The recommended level of support for record_representation_clauses should be followed. 22.b/2

NOTE If no component_clause is given for a component, then the choice of the storage place for the component is left to the implementation. If component_clauses are given for all components, the record_representation_clause completely specifies the representation of the type and will be obeyed exactly by the implementation. 23

Ramification: The visibility rules prevent the name of a component of the type from appearing in a record_representation_clause at any place *except* for the *component*_local_name of a component_clause. However, since the record_representation_clause is part of the declarative region of the type declaration, the component names hide outer homographs throughout. 23.a

{*8652/0009*} {*AI95-00137-01*} A record_representation_clause cannot be given for a protected type, even though protected types, like record types, have components. The primary reason for this rule is that there is likely to be too much dope in a protected type — entry queues, bit maps for barrier values, etc. In order to control the representation of the user-defined components, simply declare a record type, give it a record_representation_clause, and give the protected type one component whose type is the record type. Alternatively, if the protected object is protecting something like a device register, it makes more sense to keep the thing being protected outside the protected object (possibly with a pointer to it in the protected object), in order to keep implementation-defined components out of the way. 23.b/1

Examples

Example of specifying the layout of a record type: 24

```
Word : constant := 4;  -- storage element is byte, 4 bytes per word
```
25

```
type State         is (A,M,W,P);
type Mode          is (Fix, Dec, Exp, Signif);
```
26

```
{AI12-0178-1} type Byte_Mask     is array (0..7)  of Boolean with Component_Size =>
1;
type State_Mask    is array (State) of Boolean with Component_Size => 1;
type Mode_Mask     is array (Mode)  of Boolean with Component_Size => 1;
```
27/5

```
type Program_Status_Word is
  record
      System_Mask        : Byte_Mask;
      Protection_Key     : Integer range 0 .. 3;
      Machine_State      : State_Mask;
      Interrupt_Cause    : Interruption_Code;
      Ilc                : Integer range 0 .. 3;
      Cc                 : Integer range 0 .. 3;
      Program_Mask       : Mode_Mask;
      Inst_Address       : Address;
end record;
```
28

```
for Program_Status_Word use
  record
      System_Mask      at 0*Word range 0  .. 7;
      Protection_Key   at 0*Word range 10 .. 11; -- bits 8,9 unused
      Machine_State    at 0*Word range 12 .. 15;
      Interrupt_Cause  at 0*Word range 16 .. 31;
      Ilc              at 1*Word range 0  .. 1;  -- second word
      Cc               at 1*Word range 2  .. 3;
      Program_Mask     at 1*Word range 4  .. 7;
      Inst_Address     at 1*Word range 8  .. 31;
  end record;
```
29

```
for Program_Status_Word'Size use 8*System.Storage_Unit;
for Program_Status_Word'Alignment use 8;
```
30

{*AI12-0452-1*} The record_representation_clause defines the record layout. The Size clause guarantees that (at least) eight storage elements are used for objects of the type. The Alignment clause guarantees that aliased, imported, or exported objects of the type will have addresses divisible by eight. 31/5

Wording Changes from Ada 83

The alignment_clause has been renamed to mod_clause and moved to Annex J, "Obsolescent Features". 32.a

We have clarified that implementation-defined component names have to be in the form of an attribute_reference of a component or of the first subtype itself; surely Ada 83 did not intend to allow arbitrary identifiers. 32.b

32.c The RM83-13.4(7) wording incorrectly allows components in nonvariant records to overlap. We have corrected that oversight.

Incompatibilities With Ada 95

32.d/2 {*AI95-00133-01*} **Amendment Correction:** The meaning of a record_representation_clause for the nondefault bit order is now clearly defined. Thus, such clauses can be portably written. In order to do that though, the equivalence of bit 1 in word 1 to bit 9 in word 0 (for a machine with Storage_Unit = 8) had to be dropped for the nondefault bit order. Any record_representation_clauses which depends on that equivalence will break (although such code would imply a noncontiguous representation for a component, and it seems unlikely that compilers were supporting that anyway).

Extensions to Ada 95

32.e/2 {*AI95-00436-01*} **Amendment Correction:** The undocumented (and likely unintentional) incompatibility with Ada 83 caused by not allowing record_representation_clauses on limited record types is removed.

Extensions to Ada 2012

32.f/5 {*AI12-0213-1*} The local_name following **end record** is new.

13.5.2 Storage Place Attributes

Static Semantics

1 For a component C of a composite, non-array object R, the *storage place attributes* are defined:

1.a **Ramification:** The storage place attributes are not (individually) specifiable, but the user may control their values by giving a record_representation_clause.

2/2 R.C'Position

{*AI95-00133-01*} If the nondefault bit ordering applies to the composite type, and if a component_clause specifies the placement of C, denotes the value given for the position of the component_clause; otherwise, denotes the same value as R.C'Address – R'Address. The value of this attribute is of the type *universal_integer*.

2.a/2 **Ramification:** {*AI95-00133-01*} Thus, for the default bit order, R.C'Position is the offset of C in storage elements from the beginning of the object, where the first storage element of an object is numbered zero. R'Address + R.C'Position = R.C'Address. For record extensions, the offset is not measured from the beginning of the extension part, but from the beginning of the whole object, as usual.

2.b In “R.C'Address – R'Address”, the "–" operator is the one in System.Storage_Elements that takes two Addresses and returns a Storage_Offset.

3/2 R.C'First_Bit

{*AI95-00133-01*} If the nondefault bit ordering applies to the composite type, and if a component_clause specifies the placement of C, denotes the value given for the first_bit of the component_clause; otherwise, denotes the offset, from the start of the first of the storage elements occupied by C, of the first bit occupied by C. This offset is measured in bits. The first bit of a storage element is numbered zero. The value of this attribute is of the type *universal_integer*.

4/2 R.C'Last_Bit

{*AI95-00133-01*} If the nondefault bit ordering applies to the composite type, and if a component_clause specifies the placement of C, denotes the value given for the last_bit of the component_clause; otherwise, denotes the offset, from the start of the first of the storage elements occupied by C, of the last bit occupied by C. This offset is measured in bits. The value of this attribute is of the type *universal_integer*.

4.a/2 **Ramification:** {*AI95-00114-01*} The ordering of bits in a storage element is defined in 13.5.3, “Bit Ordering”.

4.b R.C'Size = R.C'Last_Bit – R.C'First_Bit + 1. (Unless the implementation chooses an indirection representation.)

4.c If a component_clause applies to a component, then that component will be at the same relative storage place in all objects of the type. Otherwise, there is no such requirement.

Implementation Advice

If a component is represented using some form of pointer (such as an offset) to the actual data of the component, and this data is contiguous with the rest of the object, then the storage place attributes should reflect the place of the actual data, not the pointer. If a component is allocated discontiguously from the rest of the object, then a warning should be generated upon reference to one of its storage place attributes. 5

Reason: For discontiguous components, these attributes make no sense. For example, an implementation might allocate dynamic-sized components on the heap. For another example, an implementation might allocate the discriminants separately from the other components, so that multiple objects of the same subtype can share discriminants. Such representations cannot happen if there is a component_clause for that component. 5.a

Implementation Advice: If a component is represented using a pointer to the actual data of the component which is contiguous with the rest of the object, then the storage place attributes should reflect the place of the actual data. If a component is allocated discontiguously from the rest of the object, then a warning should be generated upon reference to one of its storage place attributes. 5.b/2

Incompatibilities With Ada 95

{*AI95-00133-01*} **Amendment Correction:** The meaning of the storage place attributes for the nondefault bit order is now clearly defined, and can be different than that given by strictly following the Ada 95 wording. Any code which depends on the original Ada 95 values for a type using the nondefault bit order where they are different will break. 5.c/2

13.5.3 Bit Ordering

[The Bit_Order attribute specifies the interpretation of the storage place attributes.] 1

Reason: The intention is to provide uniformity in the interpretation of storage places across implementations on a particular machine by allowing the user to specify the Bit_Order. It is not intended to fully support data interoperability across different machines, although it can be used for that purpose in some situations. 1.a

{*AI95-00114-01*} We can't require all implementations on a given machine to use the same bit ordering by default; if the user cares, a Bit_Order attribute_definition_clause can be used to force all implementations to use the same bit ordering. 1.b/2

Static Semantics

A bit ordering is a method of interpreting the meaning of the storage place attributes. High_Order_First [(known in the vernacular as “big endian”)] means that the first bit of a storage element (bit 0) is the most significant bit (interpreting the sequence of bits that represent a component as an unsigned integer value). Low_Order_First [(known in the vernacular as “little endian”)] means the opposite: the first bit is the least significant. 2

{*AI12-0373-1*} For every specific record subtype S, the following representation attribute is defined: 3/5

S'Bit_Order 4

Denotes the bit ordering for the type of S. The value of this attribute is of type System.Bit_Order. Bit_Order may be specified for specific record types via an attribute_definition_clause; the expression of such a clause shall be static.

Aspect Description for Bit_Order: Order of bit numbering in a record_representation_clause. 4.a/3

If Word_Size = Storage_Unit, the default bit ordering is implementation defined. If Word_Size > Storage_Unit, the default bit ordering is the same as the ordering of storage elements in a word, when interpreted as an integer. 5

Implementation defined: If Word_Size = Storage_Unit, the default bit ordering. 5.a

Ramification: Consider machines whose Word_Size = 32, and whose Storage_Unit = 8. Assume the default bit ordering applies. On a machine with big-endian addresses, the most significant storage element of an integer is at the address of the integer. Therefore, bit zero of a storage element is the most significant bit. On a machine with little-endian addresses, the least significant storage element of an integer is at the address of the integer. Therefore, bit zero of a storage element is the least significant bit. 5.b

6 The storage place attributes of a component of a type are interpreted according to the bit ordering of the type.

6.a **Ramification:** This implies that the interpretation of the position, first_bit, and last_bit of a component_clause of a record_representation_clause obey the bit ordering given in a representation item.

Implementation Advice

7 The recommended level of support for the nondefault bit ordering is:

8/2 - {*AI95-00133-01*} The implementation should support the nondefault bit ordering in addition to the default bit ordering.

8.a/2 **Ramification:** {*AI95-00133-01*} The implementation should support both bit orderings. Implementations are required to support storage positions that cross storage element boundaries when Word_Size > Storage_Unit but the definition of the storage place attributes for the nondefault bit order ensures that such storage positions will not be split into two or three pieces. Thus, there is no significant implementation burden to supporting the nondefault bit order, given that the set of machine scalars is implementation-defined.

8.b/2 **Implementation Advice:** The recommended level of support for the nondefault bit ordering should be followed.

9/2 NOTE {*AI95-00133-01*} Bit_Order clauses make it possible to write record_representation_clauses that can be ported between machines having different bit ordering. They do not guarantee transparent exchange of data between such machines.

Extensions to Ada 83

9.a The Bit_Order attribute is new to Ada 95.

Wording Changes from Ada 95

9.b/2 {*AI95-00133-01*} We now suggest that all implementations support the nondefault bit order.

13.6 Change of Representation

1/5 {*AI05-0229-1*} {*AI12-0445-1*} [A type_conversion (see 4.6) can be used to convert between two different representations of the same array or record. To convert an array from one representation to another, two array types with matching component subtypes and convertible index types are required. If one type has Pack specified and the other does not, then explicit conversion can be used to pack or unpack an array.

2/5 {*AI12-0425-1*} {*AI12-0445-1*} To convert an untagged record from one representation to another, two record types with a common ancestor type are required. Distinct representations can then be specified for the record types, and explicit conversion between the types can be used to effect a change in representation.]

2.a/5 **Ramification:** {*AI12-0425-1*} The language does not allow implicit copying of by-reference types, so it also does not allow different representations of related by-reference types. Similarly, language rules prevent related tagged types from having different representations of the parent part. Therefore, this technique cannot be used for tagged or by-reference types.

Examples

3 *Example of change of representation:*

4
```
-- Packed_Descriptor and Descriptor are two different types
-- with identical characteristics, apart from their
-- representation
```

5
```
type Descriptor is
    record
       -- components of a descriptor
    end record;
```

6
```
type Packed_Descriptor is new Descriptor;
```

```
for Packed_Descriptor use                                                    7
    record
      -- component clauses for some or for all components
    end record;
-- Change of representation can now be accomplished by explicit type conversions:   8
D : Descriptor;                                                              9
P : Packed_Descriptor;
P := Packed_Descriptor(D);   -- pack D                                       10
D := Descriptor(P);          -- unpack P
```

13.7 The Package System

[For each implementation there is a library package called System which includes the definitions of certain configuration-dependent characteristics.] 1

Static Semantics

The following language-defined library package exists: 2

Implementation defined: The contents of the visible part of package System. 2.a/2

```
{AI95-00362-01} {AI12-0414-1} package System                                 3/5
   with Pure is

   type Name is implementation-defined-enumeration-type;                     4
   System_Name : constant Name := implementation-defined;

   -- System-Dependent Named Numbers:                                        5

   Min_Int               : constant := root_integer'First;                   6
   Max_Int               : constant := root_integer'Last;

   Max_Binary_Modulus    : constant := implementation-defined;               7
   Max_Nonbinary_Modulus : constant := implementation-defined;

   Max_Base_Digits       : constant := root_real'Digits;                     8
   Max_Digits            : constant := implementation-defined;

   Max_Mantissa          : constant := implementation-defined;               9
   Fine_Delta            : constant := implementation-defined;

   Tick                  : constant := implementation-defined;              10

   -- Storage-related Declarations:                                         11

   type Address is implementation-defined;                                  12
   Null_Address : constant Address;

   Storage_Unit : constant := implementation-defined;                       13
   Word_Size    : constant := implementation-defined * Storage_Unit;
   Memory_Size  : constant := implementation-defined;

{AI05-0229-1}     -- Address Comparison:                                     14/3
   function "<" (Left, Right : Address) return Boolean
      with Convention => Intrinsic;
   function "<="(Left, Right : Address) return Boolean
      with Convention => Intrinsic;
   function ">" (Left, Right : Address) return Boolean
      with Convention => Intrinsic;
   function ">="(Left, Right : Address) return Boolean
      with Convention => Intrinsic;
   function "=" (Left, Right : Address) return Boolean
      with Convention => Intrinsic;
-- function "/=" (Left, Right : Address) return Boolean;
   -- "/=" is implicitly defined

{AI95-00221-01}     -- Other System-Dependent Declarations:                  15/2
   type Bit_Order is (High_Order_First, Low_Order_First);
   Default_Bit_Order : constant Bit_Order := implementation-defined;
```

16
```
   -- Priority-related declarations (see D.1):
   subtype Any_Priority is Integer range implementation-defined;
   subtype Priority is Any_Priority range Any_Priority'First ..
             implementation-defined;
   subtype Interrupt_Priority is Any_Priority range Priority'Last+1 ..
             Any_Priority'Last;
```

17
```
   Default_Priority : constant Priority :=
             (Priority'First + Priority'Last)/2;
```

18
```
private
   ... -- not specified by the language
end System;
```

19 Name is an enumeration subtype. Values of type Name are the names of alternative machine configurations handled by the implementation. System_Name represents the current machine configuration.

20 The named numbers Fine_Delta and Tick are of the type *universal_real*; the others are of the type *universal_integer*.

21 The meanings of the named numbers are:

22 [Min_Int The smallest (most negative) value allowed for the expressions of a signed_integer_type_definition.

23 Max_Int The largest (most positive) value allowed for the expressions of a signed_integer_type_definition.

24 Max_Binary_Modulus
A power of two such that it, and all lesser positive powers of two, are allowed as the modulus of a modular_type_definition.

25 Max_Nonbinary_Modulus
A value such that it, and all lesser positive integers, are allowed as the modulus of a modular_type_definition.

25.a **Ramification:** There is no requirement that Max_Nonbinary_Modulus be less than or equal to Max_Binary_Modulus, although that's what makes most sense. On a typical 32-bit machine, for example, Max_Binary_Modulus will be 2**32 and Max_Nonbinary_Modulus will be 2**31, because supporting nonbinary moduli in above 2**31 causes implementation difficulties.

26 Max_Base_Digits
The largest value allowed for the requested decimal precision in a floating_point_definition.

27 Max_Digits
The largest value allowed for the requested decimal precision in a floating_point_definition that has no real_range_specification. Max_Digits is less than or equal to Max_Base_Digits.

28 Max_Mantissa
The largest possible number of binary digits in the mantissa of machine numbers of a user-defined ordinary fixed point type. (The mantissa is defined in Annex G.)

29 Fine_Delta
The smallest delta allowed in an ordinary_fixed_point_definition that has the real_range_specification **range** –1.0 .. 1.0.]

30 Tick A period in seconds approximating the real time interval during which the value of Calendar.Clock remains constant.

30.a **Ramification:** There is no required relationship between System.Tick and Duration'Small, other than the one described here.

30.b The inaccuracy of the delay_statement has no relation to Tick. In particular, it is possible that the clock used for the delay_statement is less accurate than Calendar.Clock.

We considered making Tick a run-time-determined quantity, to allow for easier configurability. However, this would not be upward compatible, and the desired configurability can be achieved using functionality defined in Annex D, “Real-Time Systems”. 30.c

Storage_Unit 31
The number of bits per storage element.

Word_Size 32
The number of bits per word.

Memory_Size 33
An implementation-defined value [that is intended to reflect the memory size of the configuration in storage elements.]

Discussion: It is unspecified whether this refers to the size of the address space, the amount of physical memory on the machine, or perhaps some other interpretation of “memory size”. In any case, the value has to be given by a static expression, even though the amount of memory on many modern machines is a dynamic quantity in several ways. Thus, Memory_Size is not very useful. 33.a

{*AI95-00161-01*} Address is a definite, nonlimited type with preelaborable initialization (see 10.2.1). Address represents machine addresses capable of addressing individual storage elements. Null_Address is an address that is distinct from the address of any object or program unit. 34/2

Ramification: The implementation has to ensure that there is at least one address that nothing will be allocated to; Null_Address will be one such address. 34.a

Ramification: Address is the type of the result of the attribute Address. 34.b

Reason: Address is required to be nonlimited and definite because it is important to be able to assign addresses, and to declare uninitialized address variables. 34.c

Ramification: {*AI95-00161-01*} {*AI12-0417-1*} If System.Address is defined as a private type (as suggested below), it might be necessary to add the Preelaborable_Initialization aspect to the declaration of Address in order that it have preelaborable initialization as required. 34.d/5

{*AI95-00221-01*} Default_Bit_Order shall be a static constant. See 13.5.3 for an explanation of Bit_Order and Default_Bit_Order. 35/2

Implementation Permissions

{*AI95-00362-01*} An implementation may add additional implementation-defined declarations to package System and its children. [However, it is usually better for the implementation to provide additional functionality via implementation-defined children of System.] 36/2

Ramification: The declarations in package System and its children can be implicit. For example, since Address is not limited, the predefined "=" and "/=" operations are probably sufficient. However, the implementation is not *required* to use the predefined "=". 36.a

Implementation Advice

Address should be a private type. 37

Reason: This promotes uniformity by avoiding having implementation-defined predefined operations for the type. We don't require it, because implementations may want to stick with what they have. 37.a

Implementation Advice: Type System.Address should be a private type. 37.a.1/2

Implementation Note: It is not necessary for Address to be able to point at individual bits within a storage element. Nor is it necessary for it to be able to point at machine registers. It is intended as a memory address that matches the hardware's notion of an address. 37.b

The representation of the **null** value of a general access type should be the same as that of Null_Address; instantiations of Unchecked_Conversion should work accordingly. If the implementation supports interfaces to other languages, the representation of the **null** value of a general access type should be the same as in those other languages, if appropriate. 37.c

Note that the children of the Interfaces package will generally provide foreign-language-specific null values where appropriate. See UI-0065 regarding Null_Address. 37.d

NOTE There are also some language-defined child packages of System defined elsewhere. 38

Extensions to Ada 83

38.a.1/1 The declarations Max_Binary_Modulus, Max_Nonbinary_Modulus, Max_Base_Digits, Null_Address, Word_Size, Bit_Order, Default_Bit_Order, Any_Priority, Interrupt_Priority, and Default_Priority are added to System in Ada 95. The presence of ordering operators for type Address is also guaranteed (the existence of these depends on the definition of Address in an Ada 83 implementation). We do not list these as incompatibilities, as the contents of System can vary between implementations anyway; thus a program that depends on the contents of System (by using **use** `System;` for example) is already at risk of being incompatible when moved between Ada implementations.

Wording Changes from Ada 83

38.a Much of the content of System is standardized, to provide more uniformity across implementations. Implementations can still add their own declarations to System, but are encouraged to do so via children of System.

38.b Some of the named numbers are defined more explicitly in terms of the standard numeric types.

38.c The pragmas System_Name, Storage_Unit, and Memory_Size are no longer defined by the language. However, the corresponding declarations in package System still exist. Existing implementations may continue to support the three pragmas as implementation-defined pragmas, if they so desire.

38.d Priority semantics, including subtype Priority, have been moved to the Real Time Annex.

Extensions to Ada 95

38.e/2 {*AI95-00161-01*} **Amendment Correction:** Type Address is defined to have preelaborable initialization, so that it can be used without restriction in preelaborated units. (If Address is defined to be a private type, as suggested by the Implementation Advice, in Ada 95 it cannot be used in some contexts in a preelaborated units. This is an unnecessary portability issue.)

38.f/2 {*AI95-00221-01*} **Amendment Correction:** Default_Bit_Order is now a static constant.

38.g/2 {*AI95-00362-01*} Package System is now Pure, so it can be portably used in more places. (Ada 95 allowed it to be Pure, but did not require that.)

13.7.1 The Package System.Storage_Elements

Static Semantics

1 The following language-defined library package exists:

2/5
```
{AI95-00362-01} {AI12-0399-1} package System.Storage_Elements
   with Pure is

3     type Storage_Offset is range implementation-defined;

4     subtype Storage_Count is Storage_Offset range 0..Storage_Offset'Last;

5     type Storage_Element is mod implementation-defined;
      for Storage_Element'Size use Storage_Unit;
      type Storage_Array is array
        (Storage_Offset range <>) of aliased Storage_Element;
      for Storage_Array'Component_Size use Storage_Unit;

6     -- Address Arithmetic:

7/3   {AI05-0229-1}    function "+"(Left : Address; Right : Storage_Offset) return
      Address
            with Convention => Intrinsic;
      function "+"(Left : Storage_Offset; Right : Address) return Address
         with Convention => Intrinsic;
      function "-"(Left : Address; Right : Storage_Offset) return Address
         with Convention => Intrinsic;
      function "-"(Left, Right : Address) return Storage_Offset
         with Convention => Intrinsic;

8/3   {AI05-0229-1}    function "mod"(Left : Address; Right : Storage_Offset)
            return Storage_Offset
               with Convention => Intrinsic;

9     -- Conversion to/from integers:
```

```
{AI05-0229-1}    type Integer_Address is implementation-defined;
   function To_Address(Value : Integer_Address) return Address
      with Convention => Intrinsic;
   function To_Integer(Value : Address) return Integer_Address
      with Convention => Intrinsic;
{AI05-0229-1} end System.Storage_Elements;
```

10/3

11/3

Reason: {*AI05-0229-1*} The Convention aspects imply that the attribute Access is not allowed for those operations. 11.a/3

The **mod** function is needed so that the definition of Alignment makes sense. 11.b

Implementation defined: The range of Storage_Elements.Storage_Offset, the modulus of Storage_Elements.-Storage_Element, and the declaration of Storage_Elements.Integer_Address. 11.c/2

Storage_Element represents a storage element. Storage_Offset represents an offset in storage elements. Storage_Count represents a number of storage elements. Storage_Array represents a contiguous sequence of storage elements. 12

Reason: The index subtype of Storage_Array is Storage_Offset because we wish to allow maximum flexibility. Most Storage_Arrays will probably have a lower bound of 0 or 1, but other lower bounds, including negative ones, make sense in some situations. 12.a

This paragraph was deleted.{*AI95-00114-01*} 12.b/2

Integer_Address is a [(signed or modular)] integer subtype. To_Address and To_Integer convert back and forth between this type and Address. 13

Implementation Requirements

Storage_Offset'Last shall be greater than or equal to Integer'Last or the largest possible storage offset, whichever is smaller. Storage_Offset'First shall be <= (–Storage_Offset'Last). 14

Paragraph 15 was deleted.

Implementation Advice

Operations in System and its children should reflect the target environment semantics as closely as is reasonable. For example, on most machines, it makes sense for address arithmetic to “wrap around”. Operations that do not make sense should raise Program_Error. 16

Implementation Advice: Operations in System and its children should reflect the target environment; operations that do not make sense should raise Program_Error. 16.a.1/2

Discussion: For example, on a segmented architecture, X < Y might raise Program_Error if X and Y do not point at the same segment (assuming segments are unordered). Similarly, on a segmented architecture, the conversions between Integer_Address and Address might not make sense for some values, and so might raise Program_Error. 16.a

Reason: We considered making Storage_Element a private type. However, it is better to declare it as a modular type in the visible part, since code that uses it is already low level, and might as well have access to the underlying representation. We also considered allowing Storage_Element to be any integer type, signed integer or modular, but it is better to have uniformity across implementations in this regard, and viewing storage elements as unsigned seemed to make the most sense. 16.b

Implementation Note: To_Address is intended for use in Address clauses. Implementations should overload To_Address if appropriate. For example, on a segmented architecture, it might make sense to have a record type representing a segment/offset pair, and have a To_Address conversion that converts from that record type to type Address. 16.c

Extensions to Ada 95

{*AI95-00362-01*} Package System.Storage_Elements is now Pure, so it can be portably used in more places. (Ada 95 allowed it to be Pure, but did not require that.) 16.d/2

13.7.2 The Package System.Address_To_Access_Conversions

Static Semantics

1 The following language-defined generic library package exists:

2/5
```
{AI12-0241-1} {AI12-0302-1} generic
   type Object(<>) is limited private;
package System.Address_To_Access_Conversions
   with Preelaborate, Nonblocking, Global => in out synchronized is
```

3/3
```
{AI05-0229-1}    type Object_Pointer is access all Object;
   function To_Pointer(Value : Address) return Object_Pointer
      with Convention => Intrinsic;
   function To_Address(Value : Object_Pointer) return Address
      with Convention => Intrinsic;
```

4/3
```
{AI05-0229-1} end System.Address_To_Access_Conversions;
```

5/2 {*AI95-00230-01*} The To_Pointer and To_Address subprograms convert back and forth between values of types Object_Pointer and Address. To_Pointer(X'Address) is equal to X'Unchecked_Access for any X that allows Unchecked_Access. To_Pointer(Null_Address) returns **null**. For other addresses, the behavior is unspecified. To_Address(**null**) returns Null_Address. To_Address(Y), where Y /= **null**, returns Y.**all**'Address.

5.a/3 **Discussion:** {*AI95-00114-01*} {*AI05-0005-1*} The programmer should ensure that the address passed to To_Pointer is either Null_Address, or the address of an object of type Object. (If Object is not a by-reference type, the object ought to be aliased; recall that the Address attribute is not required to provide a useful result for other objects.) Otherwise, the behavior of the program is unspecified; it might raise an exception or crash, for example.

5.b **Reason:** Unspecified is almost the same thing as erroneous; they both allow arbitrarily bad behavior. We don't say erroneous here, because the implementation might allow the address passed to To_Pointer to point at some memory that just happens to “look like” an object of type Object. That's not necessarily an error; it's just not portable. However, if the actual type passed to Object is (for example) an array type, the programmer would need to be aware of any dope that the implementation expects to exist, when passing an address that did not come from the Address attribute of an object of type Object.

5.c One might wonder why To_Pointer and To_Address are any better than unchecked conversions. The answer is that Address does not necessarily have the same representation as an access type. For example, an access value might point at the bounds of an array when an address would point at the first element. Or an access value might be an offset in words from someplace, whereas an address might be an offset in bytes from the beginning of memory.

Implementation Permissions

6 An implementation may place restrictions on instantiations of Address_To_Access_Conversions.

6.a **Ramification:** For example, if the hardware requires aligned loads and stores, then dereferencing an access value that is not properly aligned might raise an exception.

6.b For another example, if the implementation has chosen to use negative component offsets (from an access value), it might not be possible to preserve the semantics, since negative offsets from the Address are not allowed. (The Address attribute always points at “the first of the storage elements...”.) Note that while the implementation knows how to convert an access value into an address, it might not be able to do the reverse. To avoid generic contract model violations, the restriction might have to be detected at run time in some cases.

13.8 Machine Code Insertions

1 [A machine code insertion can be achieved by a call to a subprogram whose sequence_of_statements contains code_statements.]

Syntax

2 code_statement ::= qualified_expression;

A code_statement is only allowed in the handled_sequence_of_statements of a subprogram_body. If a subprogram_body contains any code_statements, then within this subprogram_body the only allowed form of statement is a code_statement (labeled or not), the only allowed declarative_items are use_clauses, and no exception_handler is allowed (comments and pragmas are allowed as usual). 3

Name Resolution Rules

The qualified_expression is expected to be of any type. 4

Legality Rules

The qualified_expression shall be of a type declared in package System.Machine_Code. 5

Ramification: This includes types declared in children of System.Machine_Code. 5.a

A code_statement shall appear only within the scope of a with_clause that mentions package System.Machine_Code. 6

Ramification: Note that this is not a note; without this rule, it would be possible to write machine code in compilation units which depend on System.Machine_Code only indirectly. 6.a

Static Semantics

The contents of the library package System.Machine_Code (if provided) are implementation defined. The meaning of code_statements is implementation defined. [Typically, each qualified_expression represents a machine instruction or assembly directive.] 7

Discussion: For example, an instruction might be a record with an Op_Code component and other components for the operands. 7.a

Implementation defined: The contents of the visible part of package System.Machine_Code, and the meaning of code_statements. 7.b

Implementation Permissions

An implementation may place restrictions on code_statements. An implementation is not required to provide package System.Machine_Code. 8

NOTE 1 {*AI12-0440-1*} An implementation can provide implementation-defined pragmas specifying register conventions and calling conventions. 9/5

NOTE 2 {*AI95-00318-02*} Machine code functions are exempt from the rule that a return statement is required. In fact, return statements are forbidden, since only code_statements are allowed. 10/2

Discussion: The idea is that the author of a machine code subprogram knows the calling conventions, and refers to parameters and results accordingly. The implementation should document where to put the result of a machine code function, for example, “Scalar results are returned in register 0”. 10.a

NOTE 3 Intrinsic subprograms (see 6.3.1, “Conformance Rules”) can also be used to achieve machine code insertions. Interface to assembly language can be achieved using the features in Annex B, “Interface to Other Languages”. 11

Examples

Example of a code statement: 12

```
{AI05-0229-1} M : Mask;                                                    13/3
procedure Set_Mask
  with Inline;

procedure Set_Mask is                                                      14
  use System.Machine_Code; -- assume "with System.Machine_Code;" appears somewhere above
begin
  SI_Format'(Code => SSM, B => M'Base_Reg, D => M'Disp);
  -- Base_Reg and Disp are implementation-defined attributes
end Set_Mask;
```

Extensions to Ada 83

14.a Machine code functions are allowed in Ada 95; in Ada 83, only procedures were allowed.

Wording Changes from Ada 83

14.b The syntax for code_statement is changed to say "qualified_expression" instead of "subtype_mark'record_aggregate". Requiring the type of each instruction to be a record type is overspecification.

13.9 Unchecked Type Conversions

1 [An unchecked type conversion can be achieved by a call to an instance of the generic function Unchecked_Conversion.]

Static Semantics

2 The following language-defined generic library function exists:

3/5
```
{AI05-0229-1} {AI12-0241-1} generic
   type Source(<>) is limited private;
   type Target(<>) is limited private;
function Ada.Unchecked_Conversion(S : Source) return Target
   with Pure, Nonblocking, Convention => Intrinsic;
```

3.a/3 **Reason:** {*AI05-0229-1*} The aspect Convention implies that the attribute Access is not allowed for instances of Unchecked_Conversion.

Dynamic Semantics

4 The size of the formal parameter S in an instance of Unchecked_Conversion is that of its subtype. [This is the actual subtype passed to Source, except when the actual is an unconstrained composite subtype, in which case the subtype is constrained by the bounds or discriminants of the value of the actual expression passed to S.]

5 If all of the following are true, the effect of an unchecked conversion is to return the value of an object of the target subtype whose representation is the same as that of the source object S:

6 • S'Size = Target'Size.

6.a **Ramification:** Note that there is no requirement that the Sizes be known at compile time.

7/3 • {*AI05-0078-1*} S'Alignment is a multiple of Target'Alignment or Target'Alignment is zero.

8 • The target subtype is not an unconstrained composite subtype.

9 • S and the target subtype both have a contiguous representation.

10 • The representation of S is a representation of an object of the target subtype.

11/2 {*AI95-00426-01*} Otherwise, if the result type is scalar, the result of the function is implementation defined, and can have an invalid representation (see 13.9.1). If the result type is nonscalar, the effect is implementation defined; in particular, the result can be abnormal (see 13.9.1).

11.a.1/2 **Implementation defined:** The result of unchecked conversion for instances with scalar result types whose result is not defined by the language.

11.a/2 **Implementation defined:** The effect of unchecked conversion for instances with nonscalar result types whose effect is not defined by the language.

11.a.1/2 **Reason:** {*AI95-00426-01*} Note the difference between these sentences; the first only says that the bits returned are implementation defined, while the latter allows any effect. The difference is because scalar objects should never be abnormal unless their assignment was disrupted or if they are a subcomponent of an abnormal composite object. Neither exception applies to instances of Unchecked_Conversion.

11.a.2/2 **Ramification:** {*AI95-00426-01*} Whenever unchecked conversions are used, it is the programmer's responsibility to ensure that these conversions maintain the properties that are guaranteed by the language for objects of the target type. For nonscalar types, this requires the user to understand the underlying run-time model of the implementation. The

execution of a program that violates these properties by means of unchecked conversions returning a nonscalar type is erroneous. Properties of scalar types can be checked by using the Valid attribute (see 13.9.2); programs can avoid violating properties of the type (and erroneous execution) by careful use of this attribute.

An instance of Unchecked_Conversion can be applied to an object of a private type, assuming the implementation allows it. 11.b

Implementation Permissions

An implementation may return the result of an unchecked conversion by reference, if the Source type is not a by-copy type. [In this case, the result of the unchecked conversion represents simply a different (read-only) view of the operand of the conversion.] 12

Ramification: In other words, the result object of a call on an instance of Unchecked_Conversion can occupy the same storage as the formal parameter S. 12.a

An implementation may place restrictions on Unchecked_Conversion. 13

Ramification: For example, an instantiation of Unchecked_Conversion for types for which unchecked conversion doesn't make sense may be disallowed. 13.a

Implementation Advice

{*AI95-00051-02*} Since the Size of an array object generally does not include its bounds, the bounds should not be part of the converted data. 14/2

Implementation Advice: Since the Size of an array object generally does not include its bounds, the bounds should not be part of the converted data in an instance of Unchecked_Conversion. 14.a.1/2

Ramification: On the other hand, we have no advice to offer about discriminants and tag fields. 14.a

The implementation should not generate unnecessary runtime checks to ensure that the representation of S is a representation of the target type. It should take advantage of the permission to return by reference when possible. Restrictions on unchecked conversions should be avoided unless required by the target environment. 15

Implementation Advice: There should not be unnecessary runtime checks on the result of an Unchecked_Conversion; the result should be returned by reference when possible. Restrictions on Unchecked_Conversions should be avoided. 15.a.1/2

Implementation Note: As an example of an unnecessary runtime check, consider a record type with gaps between components. The compiler might assume that such gaps are always zero bits. If a value is produced that does not obey that assumption, then the program might misbehave. The implementation should not generate extra code to check for zero bits (except, perhaps, in a special error-checking mode). 15.a

The recommended level of support for unchecked conversions is: 16

- {*AI05-0299-1*} Unchecked conversions should be supported and should be reversible in the cases where this subclause defines the result. To enable meaningful use of unchecked conversion, a contiguous representation should be used for elementary subtypes, for statically constrained array subtypes whose component subtype is one of the subtypes described in this paragraph, and for record subtypes without discriminants whose component subtypes are described in this paragraph. 17/3

Implementation Advice: The recommended level of support for Unchecked_Conversion should be followed. 17.a/2

Wording Changes from Ada 95

{*AI95-00051-02*} The implementation advice about the size of array objects was moved to 13.3 so that all of the advice about Size is in one place. 17.b/2

{*AI95-00426-01*} Clarified that the result of Unchecked_Conversion for scalar types can be invalid, but not abnormal. 17.c/2

Wording Changes from Ada 2005

{*AI05-0078-1*} **Correction:** Relaxed the alignment requirement slightly, giving a defined result in more cases. 17.d/3

13.9.1 Data Validity

1 Certain actions that can potentially lead to erroneous execution are not directly erroneous, but instead can cause objects to become *abnormal*. Subsequent uses of abnormal objects can be erroneous.

2 A scalar object can have an *invalid representation*, which means that the object's representation does not represent any value of the object's subtype. The primary cause of invalid representations is uninitialized variables.

3 Abnormal objects and invalid representations are explained in this subclause.

Dynamic Semantics

4 When an object is first created, and any explicit or default initializations have been performed, the object and all of its parts are in the *normal* state. Subsequent operations generally leave them normal. However, an object or part of an object can become *abnormal* in the following ways:

5 • An assignment to the object is disrupted due to an abort (see 9.8) or due to the failure of a language-defined check (see 11.6).

6/2 • {*AI95-00426-01*} The object is not scalar, and is passed to an **in out** or **out** parameter of an imported procedure, the Read procedure of an instance of Sequential_IO, Direct_IO, or Storage_IO, or the stream attribute T'Read, if after return from the procedure the representation of the parameter does not represent a value of the parameter's subtype.

6.1/2 • {*AI95-00426-01*} The object is the return object of a function call of a nonscalar type, and the function is an imported function, an instance of Unchecked_Conversion, or the stream attribute T'Input, if after return from the function the representation of the return object does not represent a value of the function's subtype.

6.a/2 **Discussion:** We explicitly list the routines involved in order to avoid future arguments. All possibilities are listed.

6.b/2 We did not include Stream_IO.Read in the list above. A Stream_Element should include all possible bit patterns, and thus it cannot be invalid. Therefore, the parameter will always represent a value of its subtype. By omitting this routine, we make it possible to write arbitrary I/O operations without any possibility of abnormal objects.

6.2/2 {*AI95-00426-01*} [For an imported object, it is the programmer's responsibility to ensure that the object remains in a normal state.]

6.c/2 **Proof:** This follows (and echos) the standard rule of interfacing; the programmer must ensure that Ada semantics are followed (see B.1).

7 Whether or not an object actually becomes abnormal in these cases is not specified. An abnormal object becomes normal again upon successful completion of an assignment to the object as a whole.

Erroneous Execution

8 It is erroneous to evaluate a primary that is a name denoting an abnormal object, or to evaluate a prefix that denotes an abnormal object.

8.a/2 *This paragraph was deleted.*{*AI95-00114-01*}

8.b **Ramification:** The **in out** or **out** parameter case does not apply to scalars; bad scalars are merely invalid representations, rather than abnormal, in this case.

8.c/2 **Reason:** {*AI95-00114-01*} The reason we allow access objects, and objects containing subcomponents of an access type, to become abnormal is because the correctness of an access value cannot necessarily be determined merely by looking at the bits of the object. The reason we allow scalar objects to become abnormal is that we wish to allow the compiler to optimize assuming that the value of a scalar object belongs to the object's subtype, if the compiler can prove that the object is initialized with a value that belongs to the subtype. The reason we allow composite objects to become abnormal is that such object might be represented with implicit levels of indirection; if those are corrupted, then even assigning into a component of the object, or simply asking for its Address, might have an unpredictable effect. The same is true if the discriminants have been destroyed.

Bounded (Run-Time) Errors

If the representation of a scalar object does not represent a value of the object's subtype (perhaps because the object was not initialized), the object is said to have an *invalid representation*. It is a bounded error to evaluate the value of such an object. If the error is detected, either Constraint_Error or Program_Error is raised. Otherwise, execution continues using the invalid representation. The rules of the language outside this subclause assume that all objects have valid representations. The semantics of operations on invalid representations are as follows: 9

Discussion: The AARM is more explicit about what happens when the value of the case expression is an invalid representation. 9.a

Ramification: {*AI95-00426-01*} This includes the result object of functions, including the result of Unchecked_Conversion, T'Input, and imported functions. 9.b/2

- If the representation of the object represents a value of the object's type, the value of the type is used. 10
- If the representation of the object does not represent a value of the object's type, the semantics of operations on such representations is implementation-defined, but does not by itself lead to erroneous or unpredictable execution, or to other objects becoming abnormal. 11

Implementation Note: {*AI95-00426-01*} This means that the implementation must take care not to use an invalid representation in a way that might cause erroneous execution. For instance, the exception mandated for case_statements must be raised. Array indexing must not cause memory outside of the array to be written (and usually, not read either). These cases and similar cases may require explicit checks by the implementation. 11.a/2

Erroneous Execution

{*AI95-00167-01*} {*AI05-0279-1*} A call to an imported function or an instance of Unchecked_Conversion is erroneous if the result is scalar, the result object has an invalid representation, and the result is used other than as the expression of an assignment_statement or an object_declaration, as the *object*_name of an object_renaming_declaration, or as the prefix of a Valid attribute. If such a result object is used as the source of an assignment, and the assigned value is an invalid representation for the target of the assignment, then any use of the target object prior to a further assignment to the target object, other than as the prefix of a Valid attribute reference, is erroneous. 12/3

Ramification: {*AI95-00167-01*} In a typical implementation, every bit pattern that fits in an object of a signed integer subtype will represent a value of the type, if not of the subtype. However, for an enumeration or floating point type, as well as some modular types, there are typically bit patterns that do not represent any value of the type. In such cases, the implementation ought to define the semantics of operations on the invalid representations in the obvious manner (assuming the bounded error is not detected): a given representation should be equal to itself, a representation that is in between the internal codes of two enumeration literals should behave accordingly when passed to comparison operators and membership tests, etc. We considered *requiring* such sensible behavior, but it resulted in too much arcane verbiage, and since implementations have little incentive to behave irrationally, such verbiage is not important to have. 12.a/2

{*AI95-00167-01*} If a stand-alone scalar object is initialized to a an in-range value, then the implementation can take advantage of the fact that the use of any out-of-range value has to be erroneous. Such an out-of-range value can be produced only by things like unchecked conversion, imported functions, and abnormal values caused by disruption of an assignment due to abort or to failure of a language-defined check. This depends on out-of-range values being checked before assignment (that is, checks are not optimized away unless they are proven redundant). 12.b/2

Consider the following example: 12.c

{*AI95-00167-01*} 12.d/2

```
type My_Int is range 0..99;
function Safe_Convert is new Unchecked_Conversion(My_Int, Integer);
function Unsafe_Convert is new Unchecked_Conversion(My_Int, Positive);
X : Positive := Safe_Convert(0); -- Raises Constraint_Error.
Y : Positive := Unsafe_Convert(0); -- Bounded Error, may be invalid.
B : Boolean  := Y'Valid; -- OK, B = False.
Z : Positive := Y+1; -- Erroneous to use Y.
```

{*AI95-00167-01*} {*AI95-00426-01*} The call to Unsafe_Convert is a bounded error, which might raise Constraint_Error, Program_Error, or return an invalid value. Moreover, if an exception is not raised, most uses of that invalid value (including the use of Y) cause erroneous execution. The call to Safe_Convert is not erroneous. The result object is an object of subtype Integer containing the value 0. The assignment to X is required to do a constraint check; 12.e/2

the fact that the conversion is unchecked does not obviate the need for subsequent checks required by the language rules.

12.e.1/2 {*AI95-00167-01*} {*AI95-00426-01*} The reason for delaying erroneous execution until the object is used is so that the invalid representation can be tested for validity using the Valid attribute (see 13.9.2) without causing execution to become erroneous. Note that this delay does not imply an exception will not be raised; an implementation could treat both conversions in the example in the same way and raise Constraint_Error.

12.e.2/3 {*AI05-0279-1*} The rules are defined in terms of the result object, and thus the name used to reference that object is irrelevant. That is why we don't need any special rules to describe what happens when the function result is renamed.

12.f **Implementation Note:** If an implementation wants to have a “friendly” mode, it might always assign an uninitialized scalar a default initial value that is outside the object's subtype (if there is one), and check for this value on some or all reads of the object, so as to help detect references to uninitialized scalars. Alternatively, an implementation might want to provide an “unsafe” mode where it presumed even uninitialized scalars were always within their subtype.

12.g **Ramification:** The above rules imply that it is a bounded error to apply a predefined operator to an object with a scalar subcomponent having an invalid representation, since this implies reading the value of each subcomponent. Either Program_Error or Constraint_Error is raised, or some result is produced, which if composite, might have a corresponding scalar subcomponent still with an invalid representation.

12.h Note that it is not an error to assign, convert, or pass as a parameter a composite object with an uninitialized scalar subcomponent. In the other hand, it is a (bounded) error to apply a predefined operator such as =, <, and **xor** to a composite operand with an invalid scalar subcomponent.

13/3 {*AI05-0054-2*} The dereference of an access value is erroneous if it does not designate an object of an appropriate type or a subprogram with an appropriate profile, if it designates a nonexistent object, or if it is an access-to-variable value that designates a constant object and it did not originate from an attribute_reference applied to an aliased variable view of a controlled or immutably limited object. [An access value whose dereference is erroneous can exist, for example, because of Unchecked_Deallocation, Unchecked_Access, or Unchecked_Conversion.]

13.a **Ramification:** The above mentioned Unchecked_... features are not the only causes of such access values. For example, interfacing to other languages can also cause the problem.

13.b/3 {*AI05-0054-2*} We permit the use of access-to-variable values that designate constant objects so long as they originate from an aliased variable view of a controlled or immutably limited constant, such as during the initialization of a constant (both via the “current instance” and during a call to Initialize) or during an assignment (during a call to Adjust).

14 NOTE Objects can become abnormal due to other kinds of actions that directly update the object's representation; such actions are generally considered directly erroneous, however.

Wording Changes from Ada 83

14.a In order to reduce the amount of erroneousness, we separate the concept of an undefined value into objects with invalid representation (scalars only) and abnormal objects.

14.b Reading an object with an invalid representation is a bounded error rather than erroneous; reading an abnormal object is still erroneous. In fact, the only safe thing to do to an abnormal object is to assign to the object as a whole.

Wording Changes from Ada 95

14.c/2 {*AI95-00167-01*} The description of erroneous execution for Unchecked_Conversion and imported objects was tightened up so that using the Valid attribute to test such a value is not erroneous.

14.d/2 {*AI95-00426-01*} Clarified the definition of objects that can become abnormal; made sure that all of the possibilities are included.

Wording Changes from Ada 2005

14.e/3 {*AI05-0054-2*} **Correction:** Common programming techniques such as squirreling away an access to a controlled object during initialization and using a self-referencing discriminant (the so-called “Rosen trick”) no longer are immediately erroneous if the object is declared constant, so these techniques can be used portably and safely. Practically, these techniques already worked as compilers did not take much advantage of this rule, so the impact of this change will be slight.

14.f/3 {*AI05-0279-1*} **Correction:** The description of erroneous execution for Unchecked_Conversion and imported objects was adjusted to clarify that renaming such an object is not, by itself, erroneous.

13.9.2 The Valid Attribute

The Valid attribute can be used to check the validity of data produced by unchecked conversion, input, interface to foreign languages, and the like. 1

Static Semantics

For a prefix X that denotes a scalar object [(after any implicit dereference)], the following attribute is defined: 2

X'Valid {*AI05-0153-3*} {*AI12-0071-1*} Yields True if and only if the object denoted by X is normal, has a valid representation, and then, if the preceding conditions hold, the value of X also satisfies the predicates of the nominal subtype of X. The value of this attribute is of the predefined type Boolean. 3/4

Ramification: Having checked that X'Valid is True, it is safe to read the value of X without fear of erroneous execution caused by abnormality, or a bounded error caused by an invalid representation. Such a read will produce a value in the subtype of X. 3.a

NOTE 1 Invalid data can be created in the following cases (not counting erroneous or unpredictable execution): 4

- an uninitialized scalar object, 5
- the result of an unchecked conversion, 6
- input, 7
- interface to another language (including machine code), 8
- aborting an assignment, 9
- disrupting an assignment due to the failure of a language-defined check (see 11.6), and 10
- use of an object whose Address has been specified. 11

NOTE 2 {*AI12-0071-1*} {*AI12-0440-1*} Determining whether X is normal and has a valid representation as part of the evaluation of X'Valid is not considered to include an evaluation of X; hence, it is not an error to check the validity of an object that is invalid or abnormal. Determining whether X satisfies the predicates of its nominal subtype can include an evaluation of X, but only after it has been determined that X has a valid representation. 12/5

{*AI12-0071-1*} If X is volatile, the evaluation of X'Valid is considered a read of X. 13/4

Reason: Since an implementation is not allowed to add, remove, or reorder accesses to volatile objects, we have to define X'Valid as a read so that it is implementable for most subtypes as the value of the object is required. 13.a/4

NOTE 3 {*AI95-00426-01*} {*AI12-0442-1*} The Valid attribute can be used to check the result of calling an instance of Unchecked_Conversion (or any other operation that can return invalid values). However, an exception handler is still useful because implementations are permitted to raise Constraint_Error or Program_Error if they detect the use of an invalid representation (see 13.9.1). 14/5

Ramification: If X is of an enumeration type with a representation clause, then X'Valid checks that the value of X when viewed as an integer is one of the specified internal codes. 14.a

Reason: Valid is defined only for scalar objects because the implementation and description burden would be too high for other types. For example, given a typical run-time model, it is impossible to check the validity of an access value. The same applies to composite types implemented with internal pointers. One can check the validity of a composite object by checking the validity of each of its scalar subcomponents. The user should ensure that any composite types that need to be checked for validity are represented in a way that does not involve implementation-defined components, or gaps between components. Furthermore, such types should not contain access subcomponents. 14.b

This paragraph was deleted.{*AI95-00114-01*} 14.c/2

Extensions to Ada 83

X'Valid is new in Ada 95. 14.d

Wording Changes from Ada 95

{*AI95-00426-01*} Added a note explaining that handlers for Constraint_Error and Program_Error are needed in the general case of testing for validity. (An implementation could document cases where these are not necessary, but there is no language requirement.) 14.e/2

Wording Changes from Ada 2005

14.f/3 {*AI05-0153-3*} The validity check now also includes a check of the predicate aspects (see 3.2.4), if any, of the subtype of the object.

Wording Changes from Ada 2012

14.g/4 {*AI12-0071-1*} **Corrigendum:** Updated wording of the attributes X'Valid to use the new term "satisfies the predicates" (see 3.2.4). Also updated the notes to make sense when evaluating predicates and testing validity of volatile objects.

13.10 Unchecked Access Value Creation

1 [The attribute Unchecked_Access is used to create access values in an unsafe manner — the programmer is responsible for preventing “dangling references”.]

Static Semantics

2 The following attribute is defined for a prefix X that denotes an aliased view of an object:

3 X'Unchecked_Access

All rules and semantics that apply to X'Access (see 3.10.2) apply also to X'Unchecked_Access, except that, for the purposes of accessibility rules and checks, it is as if X were declared immediately within a library package.

3.a/3 **Ramification:** {*AI05-0005-1*} We say “rules and semantics” here so that library-level accessibility applies to the value created by X'Unchecked_Access as well as to the checks needed for the attribute itself. This means that any anonymous access values that inherit the accessibility of this attribute (such as access parameters) also act as if they have library-level accessibility. We don't want the "real" accessibility of the created value re-emerging at a later point – that would create hard-to-understand bugs.

4/5 NOTE 1 {*AI12-0449-1*} This attribute is provided to support the situation where a local object is to be inserted into a global linked data structure, when the programmer knows that it will always be removed from the data structure prior to exiting the object's scope. The Access attribute would be illegal in this case (see 3.10.2).

4.a **Ramification:** The expected type for X'Unchecked_Access is as for X'Access.

4.b If an attribute_reference with Unchecked_Access is used as the actual parameter for an access parameter, an Accessibility_Check can never fail on that access parameter.

5 NOTE 2 There is no Unchecked_Access attribute for subprograms.

5.a/2 **Reason:** {*AI95-00254-01*} Such an attribute would allow unsafe “downward closures”, where an access value designating a more nested subprogram is passed to a less nested subprogram. (Anonymous access-to-subprogram parameters provide safe “downward closures”.) This requires some means of reconstructing the global environment for the more nested subprogram, so that it can do up-level references to objects. The two methods of implementing up-level references are displays and static links. If unsafe downward closures were supported, each access-to-subprogram value would have to carry the static link or display with it. We don't want to require the space and time overhead of requiring the extra information for all access-to-subprogram types, especially as including it would make interfacing to other languages (like C) harder.

5.b If desired, an instance of Unchecked_Conversion can be used to create an access value of a global access-to-subprogram type that designates a local subprogram. The semantics of using such a value are not specified by the language. In particular, it is not specified what happens if such subprograms make up-level references; even if the frame being referenced still exists, the up-level reference might go awry if the representation of a value of a global access-to-subprogram type doesn't include a static link.

13.11 Storage Management

1 [Each access-to-object type has an associated storage pool. The storage allocated by an allocator comes from the pool; instances of Unchecked_Deallocation return storage to the pool. Several access types can share the same pool.]

2/2 {*AI95-00435-01*} [A storage pool is a variable of a type in the class rooted at Root_Storage_Pool, which is an abstract limited controlled type. By default, the implementation chooses a *standard storage pool* for

each access-to-object type. The user may define new pool types, and may override the choice of pool for an access-to-object type by specifying Storage_Pool for the type.]

Ramification: By default, the implementation might choose to have a single global storage pool, which is used (by default) by all access types, which might mean that storage is reclaimed automatically only upon partition completion. Alternatively, it might choose to create a new pool at each accessibility level, which might mean that storage is reclaimed for an access type when leaving the appropriate scope. Other schemes are possible. 2.a

Term entry: storage pool object — object associated with one or more access types from which the storage for objects created by allocators of the access type(s) is obtained 2.b/5
Note: Some storage pools can be partitioned into subpools in order to support finer-grained storage management.

Legality Rules

If Storage_Pool is specified for a given access type, Storage_Size shall not be specified for it. 3

Reason: The Storage_Pool determines the Storage_Size; hence it would not make sense to specify both. Note that this rule is simplified by the fact that the aspects in question cannot be specified for derived types, nor for nonfirst subtypes, so we don't have to worry about whether, say, Storage_Pool on a derived type overrides Storage_Size on the parent type. For the same reason, “specified” means the same thing as “directly specified” here. 3.a

Static Semantics

The following language-defined library package exists: 4

```
{AI12-0235-1} with Ada.Finalization;                                                    5/5
with System.Storage_Elements;
package System.Storage_Pools
    with Pure, Nonblocking => False is

{AI95-00161-01} {AI12-0399-1}     type Root_Storage_Pool is                             6/5
        abstract new Ada.Finalization.Limited_Controlled with private
        with Preelaborable_Initialization;

    procedure Allocate(                                                                 7
      Pool : in out Root_Storage_Pool;
      Storage_Address : out Address;
      Size_In_Storage_Elements : in Storage_Elements.Storage_Count;
      Alignment : in Storage_Elements.Storage_Count) is abstract;

    procedure Deallocate(                                                               8
      Pool : in out Root_Storage_Pool;
      Storage_Address : in Address;
      Size_In_Storage_Elements : in Storage_Elements.Storage_Count;
      Alignment : in Storage_Elements.Storage_Count) is abstract;

    function Storage_Size(Pool : Root_Storage_Pool)                                     9
       return Storage_Elements.Storage_Count is abstract;

private                                                                                 10
   ... -- not specified by the language
end System.Storage_Pools;
```

Reason: The Alignment parameter is provided to Deallocate because some allocation strategies require it. If it is not needed, it can be ignored. 10.a

A *storage pool type* (or *pool type*) is a descendant of Root_Storage_Pool. The *elements* of a storage pool are the objects allocated in the pool by allocators. 11

Discussion: In most cases, an element corresponds to a single memory block allocated by Allocate. However, in some cases the implementation may choose to associate more than one memory block with a given pool element. 11.a

{*8652/0009*} {*AI95-00137-01*} {*AI95-00435-01*} For every access-to-object subtype S, the following representation attributes are defined: 12/2

S'Storage_Pool 13
: Denotes the storage pool of the type of S. The type of this attribute is Root_Storage_-Pool'Class.

14 S'Storage_Size

Yields the result of calling Storage_Size(S'Storage_Pool)[, which is intended to be a measure of the number of storage elements reserved for the pool.] The type of this attribute is *universal_integer*.

14.a **Ramification:** Storage_Size is also defined for task subtypes and objects — see 13.3.

14.b Storage_Size is not a measure of how much un-allocated space is left in the pool. That is, it includes both allocated and unallocated space. Implementations and users may provide a Storage_Available function for their pools, if so desired.

15/5 {*AI12-0319-1*} Storage_Size or Storage_Pool may be specified for a nonderived access-to-object type via an attribute_definition_clause; the name in a Storage_Pool clause shall denote a variable. If the nominal subtype of the name specified for Storage_Pool is nonblocking (see 9.5), then the primitive Allocate, Deallocate, and Storage_Size subprograms of that type shall be nonblocking. Additionally, if the pool is one that supports subpools (see 13.11.4), the primitive Default_Subpool_for_Pool, Allocate_From_Subpool, and Deallocate_Subpool subprograms shall be nonblocking.

15.a/3 **Aspect Description for Storage_Pool:** Pool of memory from which **new** will allocate for a given access type.

15.b/3 **Aspect Description for Storage_Size (access):** Sets memory size for allocations for an access type.

15.c/5 **Reason:** {*AI12-0319-1*} We need to be able to describe in contracts (especially for generic units) whether the operations of a storage pool allow blocking, and we do that with the nonblocking status of the type. We make the check when the pool is specified so we can avoid adding a special check to the declaration of a pool type – we want pool types to follow the same rules as any other type.

16/3 {*AI05-0107-1*} {*AI05-0111-3*} {*AI05-0116-1*} An allocator of a type *T* that does not support subpools allocates storage from *T*'s storage pool. If the storage pool is a user-defined object, then the storage is allocated by calling Allocate as described below. Allocators for types that support subpools are described in 13.11.4.

16.a **Ramification:** If the implementation chooses to represent the designated subtype in multiple pieces, one allocator evaluation might result in more than one call upon Allocate. In any case, allocators for the access type obtain all the required storage for an object of the designated type by calling the specified Allocate procedure.

16.b/3 *This paragraph was deleted.*{*AI05-0107-1*}

16.b.1/1 {*8652/0111*} {*AI95-00103-01*} If *D* (the designated type of *T*) includes subcomponents of other access types, they will be allocated from the storage pools for those types, even if those allocators are executed as part of the allocator of *T* (as part of the initialization of the object). For instance, an access-to-task type *TT* may allocate the data structures used to implement the task value from other storage pools. (In particular, the task stack does not necessarily need to be allocated from the storage pool for *TT*.)

17 If Storage_Pool is not specified for a type defined by an access_to_object_definition, then the implementation chooses a standard storage pool for it in an implementation-defined manner. In this case, the exception Storage_Error is raised by an allocator if there is not enough storage. It is implementation defined whether or not the implementation provides user-accessible names for the standard pool type(s).

17.a/2 *This paragraph was deleted.*

17.a.1/2 **Discussion:** The manner of choosing a storage pool is covered by a Documentation Requirement below, so it is not summarized here.

17.b **Implementation defined:** Whether or not the implementation provides user-accessible names for the standard pool type(s).

17.b.1/5 **To be honest:** {*AI12-0005-1*} **pragma** Default_Storage_Pool (see 13.11.3) can be used to specify a specific pool (or none at all) to be used with access types declared in a particular declaration list, rather than a standard storage pool.

17.b.2/5 {*AI12-0005-1*} Although there is no language-defined user-accessible name for the standard pool type(s), the default use of a standard pool may be indicated using the identifier Standard in the **pragma** Default_Storage_Pool.

17.c/2 **Ramification:** {*AI95-00230-01*} An access-to-object type defined by a derived_type_definition inherits its pool from its parent type, so all access-to-object types in the same derivation class share the same pool. Hence the "defined by an access_to_object_definition" wording above.

17.d There is no requirement that all storage pools be implemented using a contiguous block of memory (although each allocation returns a pointer to a contiguous block of memory).

{*AI12-0319-1*} {*AI12-0424-1*} The type(s) of the standard pool(s), and the primitive Allocate, Deallocate, and Storage_Size subprograms for the standard pool(s) are nonblocking. Concurrent invocations of these subprograms do not conflict with one another (see 9.10) when applied to standard storage pools. 17.1/5

Reason: We need to specify that the type is nonblocking so that an instance of Unchecked_Deallocation is nonblocking if the object type is nonblocking (as the type is used in the contract). Ada 95 did not declare standard allocation/deallocation as potentially blocking, so these things can be used in protected types, and we want that to remain true (with static checking). 17.e/5

{*AI12-0043-1*} {*AI12-0319-1*} If Storage_Size is specified for an access type *T*, an implementation-defined pool *P* is used for the type. The Storage_Size of *P* is at least that requested, and the storage for *P* is reclaimed when the master containing the declaration of the access type is left. If the implementation cannot satisfy the request, Storage_Error is raised at the freezing point of type *T*. The storage pool *P* is used only for allocators returning type *T* or other access types specified to use *T*'Storage_Pool. Storage_Error is raised by an allocator returning such a type if the storage space of *P* is exhausted (additional memory is not allocated). The type of *P*, and the primitive Allocate, Deallocate, and Storage_Size subprograms of *P* are nonblocking. 18/5

{*AI12-0043-1*} If neither Storage_Pool nor Storage_Size are specified, then the meaning of Storage_Size is implementation defined. 18.1/4

Implementation defined: The meaning of Storage_Size when neither the Storage_Size nor the Storage_Pool is specified for an access type. 18.a/2

Ramification: The Storage_Size function and attribute will return the actual size, rather than the requested size. Comments about rounding up, zero, and negative on task Storage_Size apply here, as well. See also AI83-00557, AI83-00558, and AI83-00608. 18.b

The expression in a Storage_Size clause need not be static. 18.c

The reclamation happens after the master is finalized. 18.d

Implementation Note: For a pool allocated on the stack, normal stack cut-back can accomplish the reclamation. For a library-level pool, normal partition termination actions can accomplish the reclamation. 18.e

If Storage_Pool is specified for an access type, then the specified pool is used. 19

The effect of calling Allocate and Deallocate for a standard storage pool directly (rather than implicitly via an allocator or an instance of Unchecked_Deallocation) is unspecified. 20

Ramification: For example, an allocator might put the pool element on a finalization list. If the user directly Deallocates it, instead of calling an instance of Unchecked_Deallocation, then the implementation would probably try to finalize the object upon master completion, which would be bad news. Therefore, the implementation should define such situations as erroneous. 20.a

Erroneous Execution

If Storage_Pool is specified for an access type, then if Allocate can satisfy the request, it should allocate a contiguous block of memory, and return the address of the first storage element in Storage_Address. The block should contain Size_In_Storage_Elements storage elements, and should be aligned according to Alignment. The allocated storage should not be used for any other purpose while the pool element remains in existence. If the request cannot be satisfied, then Allocate should propagate an exception [(such as Storage_Error)]. If Allocate behaves in any other manner, then the program execution is erroneous. 21

Implementation Requirements

{*AI05-0107-1*} {*AI05-0262-1*} The Allocate procedure of a user-defined storage pool object *P* may be called by the implementation only to allocate storage for a type *T* whose pool is *P*, only at the following points: 21.1/3

- During the execution of an allocator of type *T*; 21.2/3

21.a/3 **Ramification:** This includes during the evaluation of the initializing expression such as an aggregate; this is important if the initializing expression is built in place. We need to allow allocation to be deferred until the size of the object is known.

21.3/3 - During the execution of a return statement for a function whose result is built-in-place in the result of an allocator of type *T*;

21.b/3 **Reason:** We need this bullet as well as the preceding one in order that exceptions that propagate from such a call to Allocate can be handled within the return statement. We don't want to require the generation of special handling code in this unusual case, as it would add overhead to most return statements of composite types.

21.4/3 - During the execution of an assignment operation with a target of an allocated object of type *T* with a part that has an unconstrained discriminated subtype with defaults.

21.c/3 **Reason:** We allow Allocate to be called during assignment of objects with mutable parts so that mutable objects can be implemented with reallocation on assignment. (Unfortunately, the term "mutable" is only defined in the AARM, so we have to use the long-winded wording shown here.)

21.d/3 **Discussion:** Of course, explicit calls to Allocate are also allowed and are not bound by any of the rules found here.

21.5/3 {*AI05-0107-1*} {*AI05-0116-1*} {*AI05-0193-1*} {*AI05-0262-1*} {*AI05-0269-1*} For each of the calls of Allocate described above, *P* (equivalent to *T*'Storage_Pool) is passed as the Pool parameter. The Size_In_Storage_Elements parameter indicates the number of storage elements to be allocated, and is no more than *D*'Max_Size_In_Storage_Elements, where *D* is the designated subtype of *T*. The Alignment parameter is a nonzero integral multiple of *D*'Alignment if *D* is a specific type, and otherwise is a nonzero integral multiple of the alignment of the specific type identified by the tag of the object being created; it is unspecified if there is no such value. The Alignment parameter is no more than *D*'Max_Alignment_For_Allocation. The result returned in the Storage_Address parameter is used as the address of the allocated storage, which is a contiguous block of memory of Size_In_Storage_Elements storage elements. [Any exception propagated by Allocate is propagated by the construct that contained the call.]

21.e/3 **Ramification:** Note that the implementation does not turn other exceptions into Storage_Error.

21.f/3 “Nonzero integral multiple” of an alignment includes the alignment value itself, of course. The value is unspecified if the alignment of the specific type is zero.

21.6/5 {*AI05-0107-1*} {*AI12-0445-1*} The number of calls to Allocate that will be used to implement an allocator for any particular type is unspecified. The number of calls to Deallocate that will be used to implement an instance of Unchecked_Deallocation (see 13.11.2) for any particular object is the same as the number of Allocate calls for that object.

21.g/3 **Reason:** This supports objects that are allocated in one or more parts. The second sentence prevents extra or missing calls to Deallocate.

21.h/3 **To be honest:** {*AI05-0005-1*} The number of calls to Deallocate from all sources for an object always will be the same as the number of calls to Allocate from all sources for that object. However, in unusual cases, not all of those Deallocate calls may be made by an instance of Unchecked_Deallocation. Specifically, in the unusual case of assigning to an object of a mutable variant record type such that the variant changes, some of the Deallocate calls may be made by the assignment (as may some of the Allocate calls).

21.i/3 **Ramification:** We do not define the relative order of multiple calls used to deallocate the same object — that is, if the allocator allocated two pieces *x* and *y*, then an instance of Unchecked_Deallocation might deallocate *x* and then *y*, or it might deallocate *y* and then *x*.

21.7/3 {*AI05-0107-1*} The Deallocate procedure of a user-defined storage pool object *P* may be called by the implementation to deallocate storage for a type *T* whose pool is *P* only at the places when an Allocate call is allowed for *P*, during the execution of an instance of Unchecked_Deallocation for *T*, or as part of the finalization of the collection of *T*. For such a call of Deallocate, *P* (equivalent to *T*'Storage_Pool) is passed as the Pool parameter. The value of the Storage_Address parameter for a call to Deallocate is the value returned in the Storage_Address parameter of the corresponding successful call to Allocate. The values of the Size_In_Storage_Elements and Alignment parameters are the same values passed to the

corresponding Allocate call. Any exception propagated by Deallocate is propagated by the construct that contained the call.

Reason: We allow Deallocate to be called anywhere that Allocate is, in order to allow the recovery of storage from failed allocations (that is, those that raise exceptions); from extended return statements that exit via a goto, exit, or locally handled exception; and from objects that are reallocated when they are assigned. In each of these cases, we would have a storage leak if the implementation did not recover the storage (there is no way for the programmer to do it). We do not require such recovery, however, as it could be a serious performance drag on these operations. 21.j/3

Documentation Requirements

{*AI12-0445-1*} An implementation shall document the set of values that a user-defined Allocate procedure has to accept for the Alignment parameter. An implementation shall document how the standard storage pool is chosen, and how storage is allocated by standard storage pools. 22/5

This paragraph was deleted. 22.a/2

Documentation Requirement: The set of values that a user-defined Allocate procedure needs to accept for the Alignment parameter. How the standard storage pool is chosen, and how storage is allocated by standard storage pools. 22.b/2

Implementation Advice

An implementation should document any cases in which it dynamically allocates heap storage for a purpose other than the evaluation of an allocator. 23

Implementation Advice: Any cases in which heap storage is dynamically allocated other than as part of the evaluation of an allocator should be documented. 23.a.1/2

Reason: This is “Implementation Advice” because the term “heap storage” is not formally definable; therefore, it is not testable whether the implementation obeys this advice. 23.a

A default (implementation-provided) storage pool for an access-to-constant type should not have overhead to support deallocation of individual objects. 24

Implementation Advice: A default storage pool for an access-to-constant type should not have overhead to support deallocation of individual objects. 24.a.1/2

Ramification: Unchecked_Deallocation is not defined for such types. If the access-to-constant type is library-level, then no deallocation (other than at partition completion) will ever be necessary, so if the size needed by an allocator of the type is known at link-time, then the allocation should be performed statically. If, in addition, the initial value of the designated object is known at compile time, the object can be allocated to read-only memory. 24.a

Implementation Note: If the Storage_Size for an access type is specified, the storage pool should consist of a contiguous block of memory, possibly allocated on the stack. The pool should contain approximately this number of storage elements. These storage elements should be reserved at the place of the Storage_Size clause, so that allocators cannot raise Storage_Error due to running out of pool space until the appropriate number of storage elements has been used up. This approximate (possibly rounded-up) value should be used as a maximum; the implementation should not increase the size of the pool on the fly. If the Storage_Size for an access type is specified as zero, then the pool should not take up any storage space, and any allocator for the type should raise Storage_Error. 24.b

Ramification: Note that most of this is approximate, and so cannot be (portably) tested. That's why we make it an Implementation Note. There is no particular number of allocations that is guaranteed to succeed, and there is no particular number of allocations that is guaranteed to fail. 24.c

{*AI95-00230-01*} The storage pool used for an allocator of an anonymous access type should be determined as follows: 25/2

- {*AI95-00230-01*} {*AI95-00416-01*} If the allocator is defining a coextension (see 3.10.2) of an object being created by an outer allocator, then the storage pool used for the outer allocator should also be used for the coextension; 25.1/2
- {*AI95-00230-01*} For other access discriminants and access parameters, the storage pool should be created at the point of the allocator, and be reclaimed when the allocated object becomes inaccessible; 25.2/2
- {*AI05-0051-1*} If the allocator defines the result of a function with an access result, the storage pool is determined as though the allocator were in place of the call of the function. If the call is 25.3/3

the operand of a type conversion, the storage pool is that of the target access type of the conversion. If the call is itself defining the result of a function with an access result, this rule is applied recursively;

25.4/5 • {*AI95-00230-01*} {*AI12-0444-1*} Otherwise, a default storage pool should be created at the point where the anonymous access type is elaborated; such a storage pool may have no mechanism for the deallocation of individual objects.

25.a.1/2 **Implementation Advice:** Usually, a storage pool for an access discriminant or access parameter should be created at the point of an allocator, and be reclaimed when the designated object becomes inaccessible. For other anonymous access types, the pool should be created at the point where the type is elaborated and may have no mechanism for the deallocation of individual objects.

25.a/2 **Implementation Note:** {*AI95-00230-01*} For access parameters and access discriminants, the "storage pool" for an anonymous access type would not normally exist as a separate entity. Instead, the designated object of the allocator would be allocated, in the case of an access parameter, as a local aliased variable at the call site, and in the case of an access discriminant, contiguous with the object containing the discriminant. This is similar to the way storage for aggregates is typically managed.

25.b/2 {*AI95-00230-01*} For other sorts of anonymous access types, this implementation is not possible in general, as the accessibility of the anonymous access type is that of its declaration, while the allocator could be more nested. In this case, a "real" storage pool is required. Note, however, that this storage pool need not support (separate) deallocation, as it is not possible to instantiate Unchecked_Deallocation with an anonymous access type. (If deallocation is needed, the object should be allocated for a named access type and converted.) Thus, deallocation only need happen when the anonymous access type itself goes out of scope; this is similar to the case of an access-to-constant type.

26/5 NOTE 1 {*AI12-0442-1*} A user-defined storage pool type can be obtained by extending the Root_Storage_Pool type, and overriding the primitive subprograms Allocate, Deallocate, and Storage_Size. A user-defined storage pool can then be obtained by declaring an object of the type extension. The user can override Initialize and Finalize if there is any desire for nontrivial initialization and finalization for a user-defined pool type. For example, Finalize can reclaim blocks of storage that are allocated separately from the pool object itself.

27 NOTE 2 The writer of the user-defined allocation and deallocation procedures, and users of allocators for the associated access type, are responsible for dealing with any interactions with tasking. In particular:

28 • If the allocators are used in different tasks, they require mutual exclusion.

29 • If they are used inside protected objects, they cannot block.

30 • If they are used by interrupt handlers (see C.3, “Interrupt Support”), the mutual exclusion mechanism has to work properly in that context.

31 NOTE 3 The primitives Allocate, Deallocate, and Storage_Size are declared as abstract (see 3.9.3), and therefore they have to be overridden when a new (nonabstract) storage pool type is declared.

31.a **Ramification:** Note that the Storage_Pool attribute denotes an object, rather than a value, which is somewhat unusual for attributes.

31.b The calls to Allocate, Deallocate, and Storage_Size are dispatching calls — this follows from the fact that the actual parameter for Pool is T'Storage_Pool, which is of type Root_Storage_Pool'Class. In many cases (including all cases in which Storage_Pool is not specified), the compiler can determine the tag statically. However, it is possible to construct cases where it cannot.

31.c All access types in the same derivation class share the same pool, whether implementation defined or user defined. This is necessary because we allow type conversions among them (even if they are pool-specific), and we want pool-specific access values to always designate an element of the right pool.

31.d **Implementation Note:** If an access type has a standard storage pool, then the implementation doesn't actually have to follow the pool interface described here, since this would be semantically invisible. For example, the allocator could conceivably be implemented with inline code.

Examples

32 To associate an access type with a storage pool object, the user first declares a pool object of some type derived from Root_Storage_Pool. Then, the user defines its Storage_Pool attribute, as follows:

33
```
Pool_Object : Some_Storage_Pool_Type;
```

34
```
type T is access Designated;
for T'Storage_Pool use Pool_Object;
```

35/5 {*AI12-0440-1*} Another access type can be added to an existing storage pool, via:

```
for T2'Storage_Pool use T'Storage_Pool;
```

The semantics of this is implementation defined for a standard storage pool.

Reason: For example, the implementation is allowed to choose a storage pool for T that takes advantage of the fact that T is of a certain size. If T2 is not of that size, then the above will probably not work.

{*AI05-0111-3*} {*AI12-0440-1*} As usual, a derivative of Root_Storage_Pool can define additional operations. For example, consider the Mark_Release_Pool_Type defined in 13.11.6, that has two additional operations, Mark and Release, the following is a possible use:

```
{8652/0041} {AI95-00066-01} {AI05-0111-3} type Mark_Release_Pool_Type
   (Pool_Size : Storage_Elements.Storage_Count)
        is new Subpools.Root_Storage_Pool_With_Subpools with private;
           -- As defined in package MR_Pool, see 13.11.6

...

{AI05-0111-3} {AI12-0178-1} Our_Pool : Mark_Release_Pool_Type (Pool_Size => 2000);
My_Mark : Subpool_Handle; -- As declared in 13.11.6

{AI05-0111-3} type Acc is access ...;
for Acc'Storage_Pool use Our_Pool;
...

{AI05-0111-3} My_Mark := Mark(Our_Pool);
... -- Allocate objects using "new (My_Mark) Designated(...)".
Release(My_Mark); -- Finalize objects and reclaim storage.
```

Extensions to Ada 83

User-defined storage pools are new to Ada 95.

Wording Changes from Ada 83

{*AI05-0005-1*} {*AI05-0190-1*} Ada 83 originally introduced the concept called a "collection", which is similar to what we call a storage pool. All access types in the same derivation class share the same collection. Ada 95 introduces the storage pool, which is similar in that all access types in the same derivation class share the same storage pool, but other (unrelated) access types can also share the same storage pool, either by default, or as specified by the user. A collection is an amorphous grouping of objects (mainly used to describe finalization of access types); a storage pool is a more concrete concept — hence the different name.

RM83 states the erroneousness of reading or updating deallocated objects incorrectly by missing various cases.

Incompatibilities With Ada 95

{*AI95-00435-01*} **Amendment Correction:** Storage pools (and Storage_Size) are not defined for access-to-subprogram types. The original Ada 95 wording defined the attributes, but said nothing about their values. If a program uses attributes Storage_Pool or Storage_Size on an access-to-subprogram type, it will need to be corrected for Ada 2005. That's a good thing, as such a use is a bug — the concepts never were defined for such types.

Extensions to Ada 95

{*AI95-00161-01*} **Amendment Correction:** Added pragma Preelaborable_Initialization to type Root_Storage_Pool, so that extensions of it can be used to declare default-initialized objects in preelaborated units.

Wording Changes from Ada 95

{*8652/0009*} {*AI95-00137-01*} **Corrigendum:** Added wording to specify that these are representation attributes.

{*AI95-00230-01*} {*AI95-00416-01*} Added wording to clarify that an allocator for a coextension nested inside an outer allocator shares the pool with the outer allocator.

Wording Changes from Ada 2005

{*AI05-0051-1*} **Correction:** Added the missing definition of the storage pool of an allocator for an anonymous access result type.

{*AI05-0107-1*} **Correction:** Clarified when an implementation is allowed to call Allocate and Deallocate, and the requirements on such calls.

{*AI05-0111-3*} Added wording to support subpools and refer to the subpool example, see 13.11.4.

43.k/3 {*AI05-0116-1*} **Correction:** Added wording to specify that the alignment for an allocator with a class-wide designated type comes from the specific type that is allocated.

43.l/3 {*AI05-0193-1*} Added wording to allow larger alignments for calls to Allocate made by allocators, up to Max_Alignment_For_Allocation. This eases implementation in some cases.

Extensions to Ada 2012

43.m/5 {*AI12-0235-1*} Package System.Storage_Pools now has Pure categorization, so it can be mentioned for any package. Note that this does not allow storage pool objects to be shared between partitions, as these necessarily must be variables; however, this does allow a storage pool type to be shared.

Wording Changes from Ada 2012

43.n/4 {*AI12-0043-1*} **Corrigendum:** Tightened up the description of the implementation-defined pool used when Storage_Size is specified. This is not intended to change any implementation.

43.o/5 {*AI12-0319-1*} {*AI12-0424-1*} Specified that the standard pool(s), and the pool(s) used to implement a specified Storage_Size, are nonblocking and support concurrent use. This was always true (which is why this is not documented as an incompatibility), but it never was explicitly stated before. In the past, the absence of wording defining something to be potentially blocking determined if blocking was not allowed. Similarly, the absence of a statement that allocation was not allowed in multiple tasks implied that it was allowed.

13.11.1 Storage Allocation Attributes

1/3 {*AI05-0193-1*} [The Max_Size_In_Storage_Elements and Max_Alignment_For_Allocation attributes may be useful in writing user-defined pool types.]

Static Semantics

2/3 {*AI05-0193-1*} For every subtype S, the following attributes are defined:

3/5 S'Max_Size_In_Storage_Elements

{*AI95-00256-01*} {*AI95-00416-01*} {*AI05-0193-1*} {*AI12-0439-1*} Denotes the maximum value for Size_In_Storage_Elements that can be requested by the implementation via Allocate for an access type whose designated subtype is S. The value of this attribute is of type *universal_integer*.

3.a **Ramification:** If S is an unconstrained array subtype, or an unconstrained subtype with discriminants, S'Max_Size_In_Storage_Elements might be very large.

4/5 S'Max_Alignment_For_Allocation

{*AI05-0193-1*} {*AI12-0439-1*} Denotes the maximum value for Alignment that can be requested by the implementation via Allocate for an access type whose designated subtype is S. The value of this attribute is of type *universal_integer*.

5/5 {*AI05-0193-1*} {*AI12-0439-1*} For a type with access discriminants, if the implementation allocates space for a coextension in the same pool as that of the object having the access discriminant, then these attributes account for any calls on Allocate that can be performed to provide space for such coextensions.

5.a/3 **Reason:** {*AI05-0193-1*} The values of these attributes should reflect only the calls that might be made to the pool specified for an access type with designated type S. Thus, if the coextensions would normally be allocated from a different pool than the one used for the main object (that is, the Implementation Advice of 13.11 for determining the pool of an anonymous access discriminant is not followed), then these attributes should not reflect any calls on Allocate used to allocate the coextensions.

5.b/3 **Ramification:** {*AI05-0193-1*} Coextensions of coextensions of this type (and so on) are included in the values of these attributes if they are allocated from the same pool.

Wording Changes from Ada 95

5.c/2 {*AI95-00256-01*} Corrected the wording so that a fortune-telling compiler that can see the future execution of the program is not required.

Extensions to Ada 2005

{*AI05-0193-1*} The Max_Alignment_For_Allocation attribute is new. 5.d/3

13.11.2 Unchecked Storage Deallocation

[Unchecked storage deallocation of an object designated by a value of an access type is achieved by a call to an instance of the generic procedure Unchecked_Deallocation.] 1

Static Semantics

The following language-defined generic library procedure exists: 2

```
{AI05-0229-1} {AI12-0241-1} {AI12-0302-1} {AI12-0319-1} generic                    3/5
   type Object(<>) is limited private;
   type Name   is access  Object;
procedure Ada.Unchecked_Deallocation(X : in out Name)
   with Preelaborate, Nonblocking,
        Global => in out Name'Storage_Pool,
        Convention => Intrinsic;
```

Reason: {*AI05-0229-1*} The aspect Convention implies that the attribute Access is not allowed for instances of Unchecked_Deallocation. 3.a/3

Legality Rules

{*AI05-0157-1*} A call on an instance of Unchecked_Deallocation is illegal if the actual access type of the instance is a type for which the Storage_Size has been specified by a static expression with value zero or is defined by the language to be zero. In addition to the places where Legality Rules normally apply (see 12.3), this rule applies also in the private part of an instance of a generic unit. 3.1/3

Discussion: This rule is the same as the rule for allocators. We could have left the last sentence out, as a call to Unchecked_Deallocation cannot occur in a specification as it is a procedure call, but we left it for consistency and to avoid future maintenance hazards. 3.b/3

Dynamic Semantics

Given an instance of Unchecked_Deallocation declared as follows: 4

```
procedure Free is                                                                  5
    new Ada.Unchecked_Deallocation(
        object_subtype_name, access_to_variable_subtype_name);
```

Procedure Free has the following effect: 6

1. After executing Free(X), the value of X is **null**. 7
2. Free(X), when X is already equal to **null**, has no effect. 8
3. {*AI95-00416-01*} {*AI05-0107-1*} {*AI12-0445-1*} Free(X), when X is not equal to **null** first performs finalization of the object designated by X (and any coextensions of the object — see 3.10.2), as described in 7.6.1. It then deallocates the storage occupied by the object designated by X (and any coextensions). If the storage pool is a user-defined object, then the storage is deallocated by calling Deallocate as described in 13.11. There is one exception: if the object being freed contains tasks, it is unspecified whether the object is deallocated. 9/5

Ramification: {*AI05-0107-1*} Free calls only the specified Deallocate procedure to do deallocation. 9.a/3

{*AI95-00416-01*} {*AI12-0148-1*} After the finalization step of Free(X), the object designated by X, and any subcomponents (and coextensions) thereof, no longer exist; their storage can be reused for other purposes. 10/4

Bounded (Run-Time) Errors

It is a bounded error to free a discriminated, unterminated task object. The possible consequences are: 11

11.a **Reason:** This is an error because the task might refer to its discriminants, and the discriminants might be deallocated by freeing the task object.

12 • No exception is raised.

13 • Program_Error or Tasking_Error is raised at the point of the deallocation.

14 • Program_Error or Tasking_Error is raised in the task the next time it references any of the discriminants.

14.a **Implementation Note:** This last case presumes an implementation where the task references its discriminants indirectly, and the pointer is nulled out when the task object is deallocated.

15 In the first two cases, the storage for the discriminants (and for any enclosing object if it is designated by an access discriminant of the task) is not reclaimed prior to task termination.

15.a **Ramification:** The storage might never be reclaimed.

15.1/4 {*AI12-0148-1*} An access value that designates a nonexistent object is called a *dangling reference*.

15.b/4 **Discussion:** These can result from use of Unchecked_Deallocation, Unchecked_Deallocate_Subpool, and attribute Unchecked_Access. Bad results from Unchecked_Conversion and from stream-oriented attributes are abnormal by 13.9.1, which is stronger and thus takes precedence.

15.2/4 {*AI12-0148-1*} [If a dangling reference is dereferenced (implicitly or explicitly), execution is erroneous (see below).] If there is no explicit or implicit dereference, then it is a bounded error to evaluate an expression whose result is a dangling reference. If the error is detected, either Constraint_Error or Program_Error is raised. Otherwise, execution proceeds normally, but with the possibility that the access value designates some other existing object.

15.c/4 **Reason:** If a dangling reference is compared with another access value, a result of either True or False is allowed. We need to allow this so that simple implementations of access values (for instance, as a bare address) can work if the memory in question is reused. (The formal definition of access equality is that it returns True if both access values designate the same object; that can never be True if one of the values is a dangling reference, and the other is not, but both values could refer to the same memory.) Membership tests that do not involve an implicit dereference generally do not depend on the access value at all.

15.d/4 We allow Constraint_Error to be raised here so that dangling reference and null pointer checks can be combined into a single check. If different exceptions are required, then the checks have to be made separately - but there's little semantic difference (neither designate a usable object).

15.e/4 **Ramification:** If a dangling reference is assigned into an object, including being passed to a formal parameter, that object also contains a dangling reference afterwards.

15.f/4 **Discussion:** For equality and membership operations on composite types, this applies to any parts that are access types, as these operations are created based on the operations of the components (which triggers the bounded error). For other operations on composite types, the bounded error is not triggered. For instance, an assignment of a composite object with a subcomponent that is a dangling reference has to work normally; no exception can be raised, but the target object will have a subcomponent that is a dangling references, and a (direct) use of that subcomponent is again a bounded error. This is similar to the way that assignments of invalid subcomponents are handled (see 13.9.1).

Erroneous Execution

16/3 {*AI05-0033-1*} {*AI05-0262-1*} Evaluating a name that denotes a nonexistent object, or a protected subprogram or subprogram renaming whose associated object (if any) is nonexistent, is erroneous. The execution of a call to an instance of Unchecked_Deallocation is erroneous if the object was created other than by an allocator for an access type whose pool is Name'Storage_Pool.

16.a/3 **Reason:** {*AI05-0033-1*} {*AI05-0262-1*} The part about a protected subprogram is intended to cover the case of an access-to-protected-subprogram where the associated object has been deallocated. The part about a subprogram renaming is intended to cover the case of a renaming of a prefixed view where the prefix object has been deallocated, or the case of a renaming of an entry or protected subprogram where the associated task or protected object has been deallocated.

16.b/3 **Ramification:** {*AI05-0157-1*} This text does not cover the case of a name that contains a null access value, as **null** does not denote an object (rather than denoting a nonexistent object).

Implementation Advice

For a standard storage pool, Free should actually reclaim the storage. 17

Implementation Advice: For a standard storage pool, an instance of Unchecked_Deallocation should actually reclaim the storage. 17.a.1/2

Ramification: {*AI95-00114-01*} This is not a testable property, since we do not know how much storage is used by a given pool element, nor whether fragmentation can occur. 17.a/2

{*AI05-0157-1*} A call on an instance of Unchecked_Deallocation with a nonnull access value should raise Program_Error if the actual access type of the instance is a type for which the Storage_Size has been specified to be zero or is defined by the language to be zero. 17.1/3

Implementation Advice: A call on an instance of Unchecked_Deallocation with a nonnull access value should raise Program_Error if the actual access type of the instance is a type for which the Storage_Size has been specified to be zero or is defined by the language to be zero. 17.a.1/3

Discussion: If the call is not illegal (as in a generic body), we recommend that it raise Program_Error. Since the execution of this call is erroneous (any allocator from the pool will have raised Storage_Error, so the nonnull access value must have been allocated from a different pool or be a stack-allocated object), we can't require any behavior — anything at all would be a legitimate implementation. 17.b

NOTE 1 The rules here that refer to Free apply to any instance of Unchecked_Deallocation. 18

NOTE 2 Unchecked_Deallocation cannot be instantiated for an access-to-constant type. This is implied by the rules of 12.5.4. 19

Wording Changes from Ada 95

{*AI95-00416-01*} The rules for coextensions are clarified (mainly by adding that term). In theory, this reflects no change from Ada 95 (coextensions existed in Ada 95, they just didn't have a name). 19.a/2

Wording Changes from Ada 2005

{*AI05-0033-1*} **Correction:** Added a rule that using an access-to-protected-subprogram is erroneous if the associated object no longer exists. It is hard to imagine an alternative meaning here, and this has no effect on correct programs. 19.b/3

{*AI05-0107-1*} **Correction:** Moved the requirements on an implementation-generated call to Deallocate to 13.11, in order to put all of the rules associated with implementation-generated calls to Allocate and Deallocate together. 19.c/3

{*AI05-0157-1*} **Correction:** Added wording so that calling an instance of Unchecked_Deallocation is treated similarly to allocators for access types where allocators would be banned. 19.d/3

Inconsistencies With Ada 2012

{*AI12-0148-1*} **Corrigendum:** Defined a "dangling reference", and specified that a dangling reference might designate some other existing object. This allows simple implementations of access values and reuse of object memory after deallocation. In prior versions of Ada, "=" between a dangling reference and an access to an existing object has to return False, even if the existing object and the object designated by the dangling reference are allocated in the same memory. A program that depended upon that could break with this revised rule. However, as a practical matter, almost all Ada implementations use simple implementations of access types that do not meet that requirement. So such a program would not work (consistently) on most Ada implementations; thus the change shouldn't break any existing programs - it just aligns the Standard with actual practice. 19.e/4

{*AI12-0148-1*} A side effect of this change is to allow an Ada implementation to detect dangling references in more places. This does not require any Ada implementation to change, and if the implementation does change, it just means that errors will be detected earlier. 19.f/4

Wording Changes from Ada 2012

{*AI12-0148-1*} **Corrigendum:** Clarified that deallocated objects cease to exist after finalization but before Deallocate is called. This is necessary to prevent erroneous execution from being triggered by the rules in 13.11 in the time between the end of finalization and the end of the call to the instance of Unchecked_Deallocation. 19.g/4

13.11.3 Default Storage Pools

1/4 {*AI05-0229-1*} {*AI12-0003-1*} [Pragma and aspect Default_Storage_Pool specify the storage pool that will be used in the absence of an explicit specification of a storage pool or storage size for an access type.]

Syntax

2/3 {*AI05-0190-1*} {*AI05-0229-1*} The form of a pragma Default_Storage_Pool is as follows:

3/3 {*AI05-0190-1*} {*AI05-0229-1*} **pragma** Default_Storage_Pool (storage_pool_indicator);

3.1/4 {*AI05-0190-1*} {*AI12-0003-1*} storage_pool_indicator ::= *storage_pool_*name | **null** | Standard

3.2/3 {*AI05-0190-1*} A pragma Default_Storage_Pool is allowed immediately within the visible part of a package_specification, immediately within a declarative_part, or as a configuration pragma.

Name Resolution Rules

3.3/3 {*AI05-0190-1*} The *storage_pool_*name is expected to be of type Root_Storage_Pool'Class.

Legality Rules

4/3 {*AI05-0190-1*} {*AI05-0229-1*} The *storage_pool_*name shall denote a variable.

4.1/4 {*AI12-0003-1*} The Standard storage_pool_indicator is an identifier specific to a pragma (see 2.8) and does not denote any declaration. If the storage_pool_indicator is Standard, then there shall not be a declaration with defining_identifier Standard that is immediately visible at the point of the pragma, other than package Standard itself.

4.a.1/4 **Reason:** We considered having the Standard storage_pool_indicator resolve to package Standard rather than being an identifier specific to a pragma. That would eliminate the need for a special check. But it would be bizarre to have something that could resolve to either an object or a (single) package, and resolving to package Standard would imply that the standard pool is an object declared in that package. A storage pool object must be a variable (see 13.11), yet preelaborable packages depend on package Standard, which would require implementers to implement the standard storage pool with Preelaborable_Initialization, which is an unnecessary restriction.

4.a.2/4 No declaration of Standard can ever be use-visible, as the language-defined nonoverloadable definition of Standard will hide any use-visible declarations. Thus we need only concern ourselves with eliminating any possible confusion with regard to immediately visible declarations with the defining_identifier Standard.

4.2/4 {*AI05-0190-1*} {*AI12-0003-1*} If the pragma is used as a configuration pragma, the storage_pool_indicator shall be either **null** or Standard, and it defines the *default pool* to be the given storage_pool_indicator within all applicable compilation units (see 10.1.5), except within the immediate scope of another pragma Default_Storage_Pool. Otherwise, [the pragma occurs immediately within a sequence of declarations, and] it defines the default pool within the immediate scope of the pragma to be the given storage_pool_indicator, except within the immediate scope of a later pragma Default_Storage_Pool. [Thus, an inner pragma overrides an outer one.]

4.3/4 {*AI05-0190-1*} {*AI05-0262-1*} A pragma Default_Storage_Pool shall not be used as a configuration pragma that applies to a compilation unit that is within the immediate scope of another pragma Default_Storage_Pool.

4.a/3 **Reason:** This is to prevent confusion in cases like this:

4.b/3
```
package Parent is
   pragma Default_Storage_Pool(...);
   ...
end Parent;
```

```
pragma Default_Storage_Pool(...); -- Illegal!
package Parent.Child is
   ...
end Parent.Child;
```

where the Default_Storage_Pool on Parent.Child would not (if it were legal) override the one in Parent. 4.d/3

Static Semantics

{*AI05-0190-1*} {*AI05-0229-1*} {*AI12-0003-1*} The language-defined aspect Default_Storage_Pool may be specified for a generic instance; it defines the default pool for access types within an instance. . 5/4

{*AI12-0003-1*} The Default_Storage_Pool aspect may be specified as Standard, which is an identifier specific to an aspect (see 13.1.1) and defines the default pool to be Standard. In this case, there shall not be a declaration with defining_identifier Standard that is immediately visible at the point of the aspect specification, other than package Standard itself. 5.1/4

{*AI12-0003-1*} {*AI12-0445-1*} Otherwise, the expected type for the Default_Storage_Pool aspect is Root_Storage_Pool'Class and the aspect_definition shall be a name that denotes a variable. This aspect overrides any Default_Storage_Pool pragma that applies to the generic unit; if the aspect is not specified, the default pool of the instance is that defined for the generic unit. 5.2/5

Aspect Description for Default_Storage_Pool: Default storage pool for a generic instance. 5.a/3

{*AI12-0136-1*} The effect of specifying the aspect Default_Storage_Pool on an instance of a language-defined generic unit is implementation-defined. 5.3/4

Implementation defined: The effect of specifying aspect Default_Storage_Pool on an instance of a language-defined generic unit. 5.a.1/4

{*AI05-0190-1*} {*AI05-0229-1*} For nonderived access types declared in places where the default pool is defined by the pragma or aspect, their Storage_Pool or Storage_Size attribute is determined as follows, unless Storage_Pool or Storage_Size is specified for the type: 6/3

- {*AI05-0190-1*} If the default pool is **null**, the Storage_Size attribute is defined by the language to be zero. [Therefore, an allocator for such a type is illegal.] 6.1/3
- {*AI05-0190-1*} {*AI12-0003-1*} If the default pool is neither **null** nor Standard, the Storage_Pool attribute is that pool. 6.2/4

{*AI05-0190-1*} {*AI12-0003-1*} Otherwise (including when the default pool is specified as Standard), the standard storage pool is used for the type as described in 13.11. 6.3/4

Ramification: {*AI05-0190-1*} {*AI05-0229-1*} Default_Storage_Pool is the only way to specify the storage pool for an anonymous access type. 6.a/3

{*AI05-0190-1*} {*AI05-0229-1*} Note that coextensions should be allocated in the same pool (or on the stack) as the outer object (see 13.11); the Storage_Pool of the access discriminant (and hence the Default_Storage_Pool) is supposed to be ignored for coextensions. This matches the required finalization point for coextensions. 6.b/3

{*AI05-0190-1*} The default storage pool for an allocator that occurs within an instance of a generic is defined by the Default_Storage_Pool aspect of the instantiation (if specified), or by the Default_Storage_Pool pragma that applied to the generic; the Default_Storage_Pool pragma that applies to the instantiation is irrelevant. 6.b.1/3

{*AI05-0190-1*} It is possible to specify the Default_Storage_Pool aspect for an instantiation such that allocations will fail. For example, the generic unit might be expecting a pool that supports certain sizes and alignments, and the one on the instance might be more restrictive. It is the programmer's responsibility to get this right. 6.b.2/3

{*AI05-0190-1*} The semantics of the Default_Storage_Pool aspect are similar to passing a pool object as a generic formal, and putting pragma Default_Storage_Pool at the top of the generic's visible part, specifying that formal. 6.b.3/3

This paragraph was deleted.{*AI05-0229-1*} 7/3

Implementation Permissions

8/3 {*AI05-0190-1*} {*AI05-0229-1*} An object created by an allocator that is passed as the actual parameter to an access parameter may be allocated on the stack, and automatically reclaimed, regardless of the default pool.

8.a/3 **Discussion:** {*AI05-0190-1*} This matches the required finalization point for such an allocated object.

9/5 NOTE {*AI05-0190-1*} {*AI12-0440-1*} Default_Storage_Pool can be used with restrictions No_Coextensions and No_Access_Parameter_Allocators (see H.4) to ensure that all allocators use the default pool.

Wording Changes from Ada 83

9.a/3 *This paragraph was deleted.*{*AI05-0229-1*}

Incompatibilities With Ada 2005

9.b/3 {*AI05-0229-1*} Pragma Controlled has been dropped from Ada, as it has no effect in any known Ada implementations and it seems to promise capabilities not expected in Ada implementations. This is usually not an incompatibility, as the pragma merely becomes unrecognized (with a warning) and can be implemented as an implementation-defined pragma if desired. However, it is incompatible if it is (now) implemented as an implementation-defined pragma, someone used this pragma in a unit, and they also used restriction No_Implementation_Pragmas on that unit. In that case, the pragma would now violate the restriction; but use of this pragma (which does nothing) should be very rare, so this is not a significant issue.

Extensions to Ada 2005

9.c/4 {*AI05-0190-1*} {*AI12-0005-1*} The pragma Default_Storage_Pool and aspect Default_Storage_Pool are new.

Wording Changes from Ada 2005

9.d/3 {*AI05-0229-1*} The entire discussion of garbage collection (and especially that of controlled objects) is deleted. Ada 2012 provides subpools (see 13.11.4) for storage management of objects, including controlled objects, a mechanism which is much more predictable than garbage collection. Note that no version of Ada allows early finalization of controlled objects (other than via the use of Unchecked_Deallocation or Unchecked_Deallocate_Subpool), so that garbage collection of such objects would be ineffective in the standard mode anyway.

Extensions to Ada 2012

9.e/4 {*AI05-0003-1*} **Corrigendum:** The storage_pool_indicator Standard is new.

Wording Changes from Ada 2012

9.f/4 {*AI12-0136-1*} **Corrigendum:** We now explicitly say that the behavior of language-defined generic units when given the Default_Storage_Pool aspect is implementation-defined. Portable code cannot rely on such a package using a particular pool implementation.

13.11.4 Storage Subpools

1/3 {*AI05-0111-3*} This subclause defines a package to support the partitioning of a storage pool into subpools. A subpool may be specified as the default to be used for allocation from the associated storage pool, or a particular subpool may be specified as part of an allocator (see 4.8).

Static Semantics

2/3 {*AI05-0111-3*} The following language-defined library package exists:

3/5
```
{AI12-0302-1} package System.Storage_Pools.Subpools
    with Preelaborate, Global => in out synchronized is
```

4/5
```
{AI12-0356-1} {AI12-0399-1}    type Root_Storage_Pool_With_Subpools is
      abstract new Root_Storage_Pool with private
      with Preelaborable_Initialization;
```

5/5
```
{AI12-0356-1} {AI12-0399-1}    type Root_Subpool is abstract tagged limited private
      with Preelaborable_Initialization;
```

```
   type Subpool_Handle is access all Root_Subpool'Class;                                    6/3
   for Subpool_Handle'Storage_Size use 0;

   function Create_Subpool (Pool : in out Root_Storage_Pool_With_Subpools)                  7/3
      return not null Subpool_Handle is abstract;

{AI05-0252-1}     -- The following operations are intended for pool implementers:          8/3

   function Pool_of_Subpool (Subpool : not null Subpool_Handle)                             9/3
      return access Root_Storage_Pool_With_Subpools'Class;

{AI12-0302-1}     procedure Set_Pool_of_Subpool (                                         10/5
      Subpool : in not null Subpool_Handle;
      To : in out Root_Storage_Pool_With_Subpools'Class)
         with Global => overriding in out Subpool;

{AI12-0302-1}     procedure Allocate_From_Subpool (                                       11/5
      Pool : in out Root_Storage_Pool_With_Subpools;
      Storage_Address : out Address;
      Size_In_Storage_Elements : in Storage_Elements.Storage_Count;
      Alignment : in Storage_Elements.Storage_Count;
      Subpool : in not null Subpool_Handle) is abstract
         with Pre'Class => Pool_of_Subpool(Subpool) = Pool'Access,
              Global => overriding in out Subpool;

   procedure Deallocate_Subpool (                                                           12/3
      Pool : in out Root_Storage_Pool_With_Subpools;
      Subpool : in out Subpool_Handle) is abstract
         with Pre'Class => Pool_of_Subpool(Subpool) = Pool'Access;

{AI05-0298-1}     function Default_Subpool_for_Pool (                                     13/3
      Pool : in out Root_Storage_Pool_With_Subpools)
         return not null Subpool_Handle;

   overriding                                                                               14/3
   procedure Allocate (
      Pool : in out Root_Storage_Pool_With_Subpools;
      Storage_Address : out Address;
      Size_In_Storage_Elements : in Storage_Elements.Storage_Count;
      Alignment : in Storage_Elements.Storage_Count);

   overriding                                                                               15/3
   procedure Deallocate (
      Pool : in out Root_Storage_Pool_With_Subpools;
      Storage_Address : in Address;
      Size_In_Storage_Elements : in Storage_Elements.Storage_Count;
      Alignment : in Storage_Elements.Storage_Count) is null;

{AI05-0298-1}     overriding                                                              16/3
   function Storage_Size (Pool : Root_Storage_Pool_With_Subpools)
      return Storage_Elements.Storage_Count
          is (Storage_Elements.Storage_Count'Last);

private                                                                                     17/3
   ... -- not specified by the language
end System.Storage_Pools.Subpools;
```

{*AI05-0111-3*} A *subpool* is a separately reclaimable portion of a storage pool, identified by an object of type Subpool_Handle (a *subpool handle*). A subpool handle also identifies the enclosing storage pool, a *storage pool that supports subpools*, which is a storage pool whose type is descended from Root_Storage_Pool_With_Subpools. A subpool is created by calling Create_Subpool or a similar constructor; the constructor returns the subpool handle. 18/3

{*AI05-0111-3*} {*AI05-0269-1*} {*AI12-0445-1*} A *subpool object* is an object of a type descended from Root_Subpool. [Typically, subpool objects are managed by the containing storage pool; only the handles have to be exposed to clients of the storage pool. Subpool objects are designated by subpool handles, and are the run-time representation of a subpool.] 19/5

Proof: We know that subpool handles designate subpool objects because the declaration of Subpool_Handle says so. 19.a/3

20/4 {*AI05-0111-3*} {*AI12-0145-1*} Each subpool *belongs* to a single storage pool [(which will always be a pool that supports subpools)]. An access to the pool that a subpool belongs to can be obtained by calling Pool_of_Subpool with the subpool handle. Set_Pool_of_Subpool causes the subpool of the subpool handle to belong to the given pool[; this is intended to be called from subpool constructors like Create_Subpool.] Set_Pool_of_Subpool propagates Program_Error if the subpool already belongs to a pool. If Set_Pool_of_Subpool has not yet been called for a subpool, Pool_of_Subpool returns **null**.

20.a/3 **Discussion:** Pool_of_Subpool and Set_Pool_of_Subpool are provided by the Ada implementation and typically will not be overridden by the pool implementer.

21/3 {*AI05-0111-3*} When an allocator for a type whose storage pool supports subpools is evaluated, a call is made on Allocate_From_Subpool passing in a Subpool_Handle, in addition to the parameters as defined for calls on Allocate (see 13.11). The subpool designated by the *subpool_handle_*name is used, if specified in an allocator. Otherwise, Default_Subpool_for_Pool of the Pool is used to provide a subpool handle. All requirements on the Allocate procedure also apply to Allocate_from_Subpool.

21.a/3 **Discussion:** Deallocate_Subpool is expected to do whatever is needed to deallocate all of the objects contained in the subpool; it is called from Unchecked_Deallocate_Subpool (see 13.11.5).

21.b/3 Typically, the pool implementer will not override Allocate. In the canonical definition of the language, it will never be called for a pool that supports subpools (there is an Implementation Permission below that allows it to be called in certain rare cases).

Legality Rules

22/3 {*AI05-0111-3*} If a storage pool that supports subpools is specified as the Storage_Pool for an access type, the access type is called a *subpool access type*. A subpool access type shall be a pool-specific access type.

23/3 {*AI05-0111-3*} {*AI05-0252-1*} The accessibility level of a subpool access type shall not be statically deeper than that of the storage pool object. If the specified storage pool object is a storage pool that supports subpools, then the name that denotes the object shall not denote part of a formal parameter, nor shall it denote part of a dereference of a value of a non-library-level general access type. In addition to the places where Legality Rules normally apply (see 12.3), these rules also apply in the private part of an instance of a generic unit.

Dynamic Semantics

24/3 {*AI05-0111-3*} {*AI05-0252-1*} When an access type with a specified storage pool is frozen (see 13.14), if the tag of the storage pool object identifies a storage pool that supports subpools, the following checks are made:

25/3 - the name used to specify the storage pool object does not denote part of a formal parameter nor part of a dereference of a value of a non-library-level general access type; and

26/3 - the accessibility level of the access type is not deeper than that of the storage pool object.

27/3 {*AI05-0252-1*} Program_Error is raised if either of these checks fail.

27.a/3 **Reason:** This check (and its static counterpart) ensures that the type of the allocated objects exists at least as long as the storage pool object, so that the subpools are finalized (which finalizes any remaining allocated objects) before the type of the objects ceases to exist. The access type itself (and the associated collection) will cease to exist before the storage pool ceases to exist.

27.b/3 We also disallow the use of formal parameters and dereferences of non-library-level general access types when specifying a storage pool object if it supports subpools, because the "apparent" accessibility level is potentially deeper than that of the underlying object. Neither of these cases is very likely to occur in practice.

28/3 {*AI05-0111-3*} A call to Subpools.Allocate(P, Addr, Size, Align) does the following:

```
Allocate_From_Subpool                                                    29/3
  (Root_Storage_Pool_With_Subpools'Class(P),
   Addr, Size, Align,
   Subpool => Default_Subpool_for_Pool
                (Root_Storage_Pool_With_Subpools'Class(P)));
```

{*AI05-0111-3*} An allocator that allocates in a subpool raises Program_Error if the allocated object has task parts. 30/3

Reason: This is to ease implementation. We envision relaxing this restriction in a future version of Ada, once implementation experience has been gained. At this time, we are unable to come up with a set of rules for task termination that is both useful, and surely feasible to implement. 30.a/3

{*AI05-0111-3*} Unless overridden, Default_Subpool_for_Pool propagates Program_Error. 31/3

Erroneous Execution

{*AI12-0142-1*} If Allocate_From_Subpool does not meet one or more of the requirements on the Allocate procedure as given in the Erroneous Execution rules of 13.11, then the program execution is erroneous. 31.1/4

Implementation Permissions

{*AI05-0111-3*} When an allocator for a type whose storage pool is of type Root_Storage_Pool'Class is evaluated, but supports subpools, the implementation may call Allocate rather than Allocate_From_Subpool. [This will have the same effect, so long as Allocate has not been overridden.] 32/3

Reason: This ensures either of two implementation models are possible for an allocator with no subpool_specification. Note that the "supports subpools" property is not known at compile time for a pool of the class-wide type. 32.a/3

- The implementation can dispatch to Storage_Pools.Allocate. If the pool supports subpools, this will call Allocate_From_Subpool with the default subpool so long as Allocate has not been overridden. 32.b/3
- The implementation can declare Allocate_From_Subpool as a primitive of Root_Storage_Pool in the private part of Storage_Pools. This means that the Allocate_From_Subpool for Root_Storage_Pool_With_Subpools overrides that private one. The implementation can thus call the private one, which will call Allocate for non-subpool-supporting pools. The effect of this implementation does not change if Allocate is overridden for a pool that supports subpools. 32.c/3

NOTE 1 {*AI05-0111-3*} {*AI12-0442-1*} {*AI12-0447-1*} A user-defined storage pool type that supports subpools can be implemented by extending the Root_Storage_Pool_With_Subpools type, and overriding the primitive subprograms Create_Subpool, Allocate_From_Subpool, and Deallocate_Subpool. Create_Subpool is expected to call Set_Pool_Of_Subpool before returning the subpool handle. To make use of such a pool, a user can declare an object of the type extension, can use it to define the Storage_Pool attribute of one or more access types, and then can invoke Create_Subpool to obtain subpool handles associated with the pool. 33/5

NOTE 2 {*AI05-0111-3*} {*AI12-0440-1*} A user-defined storage pool type that supports subpools can define additional subpool constructors similar to Create_Subpool (these typically will have additional parameters). 34/5

NOTE 3 {*AI05-0111-3*} {*AI12-0442-1*} The pool implementor can override Default_Subpool_For_Pool if they want the pool to support a default subpool for the pool. The implementor can override Deallocate if individual object reclamation is to be supported, and can override Storage_Size if there is some limit on the total size of the storage pool. The implementor can override Initialize and Finalize if there is any desire for nontrivial initialization and finalization for the pool as a whole. For example, Finalize can reclaim blocks of storage that are allocated over and above the space occupied by the pool object itself. The pool implementor can extend the Root_Subpool type as necessary to carry additional information with each subpool provided by Create_Subpool. 35/5

Extensions to Ada 2005

{*AI05-0111-3*} {*AI05-0252-1*} Subpools and the package System.Storage_Pools.Subpools are new. 35.a/3

Wording Changes from Ada 2012

{*AI12-0142-1*} **Corrigendum**: Clarified that an incorrect implementation of Allocate_From_Subpool causes execution to become erroneous. The wording already said that the requirements of Allocate apply to Allocate_From_Subpool, so we're just confirming the consequences of violating those requirements also apply. 35.b/4

35.c/4 {*AI12-0145-1*} **Corrigendum**: Clarified that Pool_of_Subpool returns **null** if Set_Pool_of_Subpool has not been called. As that can be inferred from the definition, and all known existing implementations return **null** in this case, we document this as a wording change rather than a possible inconsistency.

13.11.5 Subpool Reclamation

1/3 {*AI05-0111-3*} A subpool may be explicitly deallocated using Unchecked_Deallocate_Subpool.

Static Semantics

2/3 {*AI05-0111-3*} The following language-defined library procedure exists:

3/5
```
{AI12-0302-1} with System.Storage_Pools.Subpools;
procedure Ada.Unchecked_Deallocate_Subpool
   (Subpool : in out System.Storage_Pools.Subpools.Subpool_Handle)
   with Global => in out all;
```

3.a/5 **Discussion:** {*AI12-0005-1*} {*AI12-0302-1*} The Global specification for this routine needs to account for the dispatching call to the user-defined Deallocate_Subpool routine. We can't use the Dispatching aspect (see H.7.1) as that requires a statically named object (we have a function call here), so we have to use **in out all** in order to allow the user-defined subprogram to do anything it needs to do.

4/3 {*AI05-0111-3*} If Subpool is **null**, a call on Unchecked_Deallocate_Subpool has no effect. Otherwise, the subpool is finalized, and Subpool is set to **null**.

5/5 {*AI05-0111-3*} {*AI12-0331-1*} Finalization of a subpool has the following effects in the given order:

6/5 *This paragraph was deleted.*{*AI12-0331-1*}

7/3 1. Any of the objects allocated from the subpool that still exist are finalized in an arbitrary order;

7.1/4 2. {*AI12-0148-1*} All of the objects allocated from the subpool cease to exist;

8/3 3. The following [dispatching] call is then made:

9/3
```
Deallocate_Subpool(Pool_of_Subpool(Subpool).all, Subpool);
```

9.1/5 4. {*AI12-0331-1*} The subpool ceases to belong to any pool.

10/3 {*AI05-0111-3*} Finalization of a Root_Storage_Pool_With_Subpools object finalizes all subpools that belong to that pool that have not yet been finalized.

10.a/3 **Discussion:** There is no need to call Unchecked_Deallocation on an object allocated in a subpool. Such objects are deallocated all at once, when Unchecked_Deallocate_Subpool is called.

10.b/3 If Unchecked_Deallocation is called, the object is finalized, and then Deallocate is called on the Pool, which typically will do nothing. If it wants to free memory, it will need some way to get from the address of the object to the subpool.

10.c/3 There is no Deallocate_From_Subpool. There is no efficient way for the implementation to determine the subpool for an arbitrary object, and if the pool implementer can determine that, they can use that as part of the implementation of Deallocate.

10.d/3 If Unchecked_Deallocation is not called (the usual case), the object will be finalized when Unchecked_Deallocate_Subpool is called.

10.e/3 If that's never called, then the object will be finalized when the Pool_With_Subpools is finalized (by permission — it might happen when the collection of the access type is finalized).

Extensions to Ada 2005

10.f/3 {*AI05-0111-3*} Unchecked_Deallocate_Subpool is new.

Wording Changes from Ada 2012

10.g/4 {*AI12-0148-1*} **Corrigendum:** Added missing wording to state that the objects cease to exist after the completion of finalization. This is formally an inconsistency (it would be possible to depend on the fact that objects finalized by Unchecked_Deallocate_Subpool still exist), but that violates every sane expectation for a procedure called "Deallocate" something.

{*AI12-0331-1*} Clarified that the steps of deallocating a subpool occur in a specific order. This shouldn't change any implementation; no implementation is going to finalize deallocated objects or implement Unchecked_Deallocate_Subpool so it is certain to raise Constraint_Error. 10.h/5

13.11.6 Storage Subpool Example

Examples

{*AI05-0111-3*} The following example is a simple but complete implementation of the classic Mark/Release pool using subpools: 1/3

```
with System.Storage_Pools.Subpools;                                                    2/3
with System.Storage_Elements;
with Ada.Unchecked_Deallocate_Subpool;
package MR_Pool is

   use System.Storage_Pools;                                                           3/3
      -- For uses of Subpools.
   use System.Storage_Elements;
      -- For uses of Storage_Count and Storage_Array.

   -- Mark and Release work in a stack fashion, and allocations are not allowed       4/3
   -- from a subpool other than the one at the top of the stack. This is also
   -- the default pool.

   subtype Subpool_Handle is Subpools.Subpool_Handle;                                  5/3

   type Mark_Release_Pool_Type (Pool_Size : Storage_Count) is new                      6/3
      Subpools.Root_Storage_Pool_With_Subpools with private;

   function Mark (Pool : in out Mark_Release_Pool_Type)                                7/3
      return not null Subpool_Handle;

   procedure Release (Subpool : in out Subpool_Handle) renames                         8/3
      Ada.Unchecked_Deallocate_Subpool;

private                                                                                9/3

   type MR_Subpool is new Subpools.Root_Subpool with record                            10/3
      Start : Storage_Count;
   end record;
   subtype Subpool_Indexes is Positive range 1 .. 10;
   type Subpool_Array is array (Subpool_Indexes) of aliased MR_Subpool;

{AI05-0298-1} {AI12-0134-1}    type Mark_Release_Pool_Type (Pool_Size :             11/4
Storage_Count) is new
      Subpools.Root_Storage_Pool_With_Subpools with record
      Storage         : Storage_Array (0 .. Pool_Size);
      Next_Allocation : Storage_Count := 0;
      Markers         : Subpool_Array;
      Current_Pool    : Subpool_Indexes := 1;
   end record;

{AI05-0298-1}    overriding                                                          12/3
   function Create_Subpool (Pool : in out Mark_Release_Pool_Type)
      return not null Subpool_Handle;

   function Mark (Pool : in out Mark_Release_Pool_Type)                                13/3
      return not null Subpool_Handle renames Create_Subpool;

   overriding                                                                          14/3
   procedure Allocate_From_Subpool (
      Pool : in out Mark_Release_Pool_Type;
      Storage_Address : out System.Address;
      Size_In_Storage_Elements : in Storage_Count;
      Alignment : in Storage_Count;
      Subpool : not null Subpool_Handle);

   overriding                                                                          15/3
   procedure Deallocate_Subpool (
      Pool : in out Mark_Release_Pool_Type;
      Subpool : in out Subpool_Handle);
```

```
16/3     {AI05-0298-1}     overriding
            function Default_Subpool_for_Pool (Pool : in out Mark_Release_Pool_Type)
               return not null Subpool_Handle;
17/3        overriding
            procedure Initialize (Pool : in out Mark_Release_Pool_Type);
18/3        -- We don't need Finalize.
19/3     end MR_Pool;
20/3     package body MR_Pool is
21/3     {AI05-0298-1}     use type Subpool_Handle;
22/3     {AI05-0298-1}     procedure Initialize (Pool : in out Mark_Release_Pool_Type) is
            -- Initialize the first default subpool.
            begin
               Pool.Markers(1).Start := 1;
               Subpools.Set_Pool_of_Subpool
                  (Pool.Markers(1)'Unchecked_Access, Pool);
            end Initialize;
23/3        function Create_Subpool (Pool : in out Mark_Release_Pool_Type)
               return not null Subpool_Handle is
               -- Mark the current allocation location.
            begin
               if Pool.Current_Pool = Subpool_Indexes'Last then
                  raise Storage_Error; -- No more subpools.
               end if;
               Pool.Current_Pool := Pool.Current_Pool + 1; -- Move to the next subpool
24/3        {AI05-0298-1}        return Result : constant not null Subpool_Handle :=
               Pool.Markers(Pool.Current_Pool)'Unchecked_Access
            do
               Pool.Markers(Pool.Current_Pool).Start := Pool.Next_Allocation;
               Subpools.Set_Pool_of_Subpool (Result, Pool);
            end return;
         end Create_Subpool;
25/3     {AI05-0298-1}     procedure Deallocate_Subpool (
            Pool : in out Mark_Release_Pool_Type;
            Subpool : in out Subpool_Handle) is
         begin
            if Subpool /= Pool.Markers(Pool.Current_Pool)'Unchecked_Access then
               raise Program_Error; -- Only the last marked subpool can be released.
            end if;
            if Pool.Current_Pool /= 1 then
               Pool.Next_Allocation := Pool.Markers(Pool.Current_Pool).Start;
               Pool.Current_Pool := Pool.Current_Pool - 1; -- Move to the previous subpool
            else -- Reinitialize the default subpool:
               Pool.Next_Allocation := 1;
               Subpools.Set_Pool_of_Subpool
                  (Pool.Markers(1)'Unchecked_Access, Pool);
            end if;
         end Deallocate_Subpool;
26/3     {AI05-0298-1}     function Default_Subpool_for_Pool (Pool : in out
      Mark_Release_Pool_Type)
            return not null Subpool_Handle is
         begin
            return Pool.Markers(Pool.Current_Pool)'Unchecked_Access;
         end Default_Subpool_for_Pool;
```

```
   procedure Allocate_From_Subpool (                                                    27/3
      Pool : in out Mark_Release_Pool_Type;
      Storage_Address : out System.Address;
      Size_In_Storage_Elements : in Storage_Count;
      Alignment : in Storage_Count;
      Subpool : not null Subpool_Handle) is
   begin
      if Subpool /= Pool.Markers(Pool.Current_Pool)'Unchecked_Access then
         raise Program_Error; -- Only the last marked subpool can be used for allocations.
      end if;
{AI12-0080-1}       -- Check for the maximum supported alignment, which is the alignment of the storage area:  28/4
      if Alignment > Pool.Storage'Alignment then
         raise Program_Error;
      end if;
      -- Correct the alignment if necessary:
      Pool.Next_Allocation := Pool.Next_Allocation +
         ((-Pool.Next_Allocation) mod Alignment);
      if Pool.Next_Allocation + Size_In_Storage_Elements >
         Pool.Pool_Size then
         raise Storage_Error; -- Out of space.
      end if;
      Storage_Address := Pool.Storage (Pool.Next_Allocation)'Address;
      Pool.Next_Allocation :=
         Pool.Next_Allocation + Size_In_Storage_Elements;
   end Allocate_From_Subpool;
end MR_Pool;                                                                        29/3
```

Wording Changes from Ada 2005

{*AI05-0111-3*} This example of subpools is new. 29.a/3

13.12 Pragma Restrictions and Pragma Profile

{*AI05-0246-1*} [A pragma Restrictions expresses the user's intent to abide by certain restrictions. A pragma Profile expresses the user's intent to abide by a set of Restrictions or other specified run-time policies. These may facilitate the construction of simpler run-time environments.] 1/3

Syntax

The form of a pragma Restrictions is as follows: 2

pragma Restrictions(restriction{, restriction}); 3

{*AI95-00381-01*} restriction ::= *restriction_*identifier 4/2
| *restriction_parameter_*identifier => restriction_parameter_argument

{*AI95-00381-01*} restriction_parameter_argument ::= name | expression 4.1/2

Name Resolution Rules

Unless otherwise specified for a particular restriction, the expression is expected to be of any integer type. 5

Legality Rules

Unless otherwise specified for a particular restriction, the expression shall be static, and its value shall be nonnegative. 6

This paragraph was deleted. 7.a/3

Paragraph 7 was deleted.

Post-Compilation Rules

8/3 {*AI05-0013-1*} A pragma Restrictions is a configuration pragma. If a pragma Restrictions applies to any compilation unit included in the partition, this may impose either (or both) of two kinds of requirements, as specified for the particular restriction:

8.1/3 - {*AI05-0013-1*} A restriction may impose requirements on some or all of the units comprising the partition. Unless otherwise specified for a particular restriction, such a requirement applies to all of the units comprising the partition and is enforced via a post-compilation check.

8.2/3 - {*AI05-0013-1*} A restriction may impose requirements on the run-time behavior of the program, as indicated by the specification of run-time behavior associated with a violation of the requirement.

8.a.1/3 **Ramification:** In this latter case, there is no post-compilation check needed for the requirement.

8.3/1 {*8652/0042*} {*AI95-00130-01*} For the purpose of checking whether a partition contains constructs that violate any restriction (unless specified otherwise for a particular restriction):

8.4/1 - {*8652/0042*} {*AI95-00130-01*} Generic instances are logically expanded at the point of instantiation;

8.5/1 - {*8652/0042*} {*AI95-00130-01*} If an object of a type is declared or allocated and not explicitly initialized, then all expressions appearing in the definition for the type and any of its ancestors are presumed to be used;

8.6/1 - {*8652/0042*} {*AI95-00130-01*} A default_expression for a formal parameter or a generic formal object is considered to be used if and only if the corresponding actual parameter is not provided in a given call or instantiation.

Implementation Permissions

8.7/3 {*AI05-0269-1*} An implementation may provide implementation-defined restrictions; the identifier for an implementation-defined restriction shall differ from those of the language-defined restrictions.

8.a.2/3 **Implementation defined:** Implementation-defined restrictions allowed in a pragma Restrictions.

9 An implementation may place limitations on the values of the expression that are supported, and limitations on the supported combinations of restrictions. The consequences of violating such limitations are implementation defined.

9.a **Implementation defined:** The consequences of violating limitations on Restrictions pragmas.

9.b **Ramification:** Such limitations may be enforced at compile time or at run time. Alternatively, the implementation is allowed to declare violations of the restrictions to be erroneous, and not enforce them at all.

9.1/1 {*8652/0042*} {*AI95-00130-01*} An implementation is permitted to omit restriction checks for code that is recognized at compile time to be unreachable and for which no code is generated.

9.2/1 {*8652/0043*} {*AI95-00190-01*} Whenever enforcement of a restriction is not required prior to execution, an implementation may nevertheless enforce the restriction prior to execution of a partition to which the restriction applies, provided that every execution of the partition would violate the restriction.

Syntax

10/3 {*AI95-00249-01*} {*AI05-0246-1*} The form of a pragma Profile is as follows:

11/3 **pragma** Profile (*profile*_identifier {, *profile*_pragma_argument_association});

Legality Rules

12/3 {*AI95-00249-01*} {*AI05-0246-1*} The *profile*_identifier shall be the name of a usage profile. The semantics of any *profile*_pragma_argument_associations are defined by the usage profile specified by the *profile*_identifier.

Static Semantics

{*AI95-00249-01*} {*AI05-0246-1*} A profile is equivalent to the set of configuration pragmas that is defined for each usage profile. 13/3

Post-Compilation Rules

{*AI95-00249-01*} A pragma Profile is a configuration pragma. There may be more than one pragma Profile for a partition. 14/3

Implementation Permissions

{*AI05-0269-1*} An implementation may provide implementation-defined usage profiles; the identifier for an implementation-defined usage profile shall differ from those of the language-defined usage profiles. 15/3

Implementation defined: Implementation-defined usage profiles allowed in a pragma Profile. 15.a.1/3

NOTE 1 {*AI95-00347-01*} Restrictions intended to facilitate the construction of efficient tasking run-time systems are defined in D.7. Restrictions intended for use when constructing high integrity systems are defined in H.4. 16/2

NOTE 2 An implementation has to enforce the restrictions in cases where enforcement is required, even if it chooses not to take advantage of the restrictions in terms of efficiency. 17

Discussion: It is not the intent that an implementation will support a different run-time system for every possible combination of restrictions. An implementation might support only two run-time systems, and document a set of restrictions that is sufficient to allow use of the more efficient and safe one. 17.a

Extensions to Ada 83

Pragma Restrictions is new to Ada 95. 17.b

Extensions to Ada 95

{*AI95-00249-01*} {*AI05-0246-1*} Pragma Profile is new; it was moved here by Ada 2012 and renamed to a "usage profile" but was otherwise unchanged. 17.c/3

Wording Changes from Ada 95

{*8652/0042*} {*AI95-00130-01*} **Corrigendum:** Corrected the wording so that restrictions are checked inside of generic instantiations and in default expressions. Since not making these checks would violate the purpose of restrictions, we are not documenting this as an incompatibility. 17.d/2

{*8652/0043*} {*AI95-00190-01*} **Corrigendum:** Added a permission that restrictions can be enforced at compile-time. While this is technically incompatible, documenting it as such would be unnecessarily alarming - there should not be any programs depending on the runtime failure of restrictions. 17.e/2

{*AI95-00381-01*} The syntax of a restriction_parameter_argument has been defined to better support restriction No_Dependence (see 13.12.1). 17.f/2

Wording Changes from Ada 2005

{*AI05-0013-1*} **Correction**: When restrictions are checked has been clarified. 17.g/3

13.12.1 Language-Defined Restrictions and Profiles

Static Semantics

{*AI95-00257-01*} The following *restriction*_identifiers are language defined (additional restrictions are defined in the Specialized Needs Annexes): 1/2

{*AI05-0241-1*} No_Implementation_Aspect_Specifications 1.1/3

There are no implementation-defined aspects specified by an aspect_specification. This restriction applies only to the current compilation or environment, not the entire partition.

Discussion: {*AI05-0241-1*} This restriction (as well as others below) applies only to the current compilation, because it is likely that the runtime (and possibly user-written low-level code) will need to use implementation-defined aspects. But a partition-wide restriction applies everywhere, including the runtime. 1.a/3

2/2 {*AI95-00257-01*} No_Implementation_Attributes

There are no implementation-defined attributes. This restriction applies only to the current compilation or environment, not the entire partition.

2.1/3 {*AI05-0246-1*} {*AI05-0269-1*} No_Implementation_Identifiers

There are no usage names that denote declarations with implementation-defined identifiers that occur within language-defined packages or instances of language-defined generic packages. Such identifiers can arise as follows:

2.2/3 - The following language-defined packages and generic packages allow implementation-defined identifiers:

2.3/3 - package System (see 13.7);

2.4/3 - package Standard (see A.1);

2.5/3 - package Ada.Command_Line (see A.15);

2.6/3 - package Interfaces.C (see B.3);

2.7/3 - package Interfaces.C.Strings (see B.3.1);

2.8/3 - package Interfaces.C.Pointers (see B.3.2);

2.9/3 - package Interfaces.COBOL (see B.4);

2.10/3 - package Interfaces.Fortran (see B.5);

2.11/3 - The following language-defined packages contain only implementation-defined identifiers:

2.12/3 - package System.Machine_Code (see 13.8);

2.13/3 - package Ada.Directories.Information (see A.16);

2.14/3 - nested Implementation packages of the Queue containers (see A.18.28-31);

2.15/3 - package Interfaces (see B.2);

2.16/3 - package Ada.Interrupts.Names (see C.3.2).

2.17/3 For package Standard, Standard.Long_Integer and Standard.Long_Float are considered language-defined identifiers, but identifiers such as Standard.Short_Short_Integer are considered implementation-defined.

2.18/3 This restriction applies only to the current compilation or environment, not the entire partition.

3/2 {*AI95-00257-01*} No_Implementation_Pragmas

There are no implementation-defined pragmas or pragma arguments. This restriction applies only to the current compilation or environment, not the entire partition.

3.1/3 {*AI05-0242-1*} No_Implementation_Units

There is no mention in the context_clause of any implementation-defined descendants of packages Ada, Interfaces, or System. This restriction applies only to the current compilation or environment, not the entire partition.

4/3 {*AI95-00368-01*} {*AI05-0229-1*} No_Obsolescent_Features

There is no use of language features defined in Annex J. It is implementation defined whether uses of the renamings of J.1 and of the pragmas of J.15 are detected by this restriction. This restriction applies only to the current compilation or environment, not the entire partition.

4.a/2 **Reason:** A user could compile a rename like

4.b/2
```
with Ada.Text_IO;
package Text_IO renames Ada.Text_IO;
```

Such a rename must not be disallowed by this restriction, nor should the compilation of such a rename be restricted by an implementation. Many implementations implement the renames of J.1 by compiling them normally; we do not want to require implementations to use a special mechanism to implement these renames. 4.c/2

{*AI05-0229-1*} The pragmas have the same functionality as the corresponding aspect (unlike the typical obsolescent feature), and rejecting them could be a significant portability problem for existing code. 4.d/3

{*AI95-00381-01*} {*AI05-0241-1*} The following *restriction_parameter_*identifiers are language defined: 5/3

{*AI95-00381-01*} No_Dependence 6/2
Specifies a library unit on which there are no semantic dependences.

{*AI05-0241-1*} No_Specification_of_Aspect 6.1/3
Identifies an aspect for which no aspect_specification, attribute_definition_clause, or pragma is given.

{*AI05-0272-1*} No_Use_Of_Attribute 6.2/3
Identifies an attribute for which no attribute_reference or attribute_definition_clause is given.

{*AI05-0272-1*} No_Use_Of_Pragma 6.3/3
Identifies a pragma which is not to be used.

{*AI12-0389-1*} No_Unrecognized_Aspects 6.4/5
There are no aspect_specifications having an unrecognized *aspect_*identifier. This restriction applies only to the current compilation or environment, not the entire partition.

Ramification: When this restriction is in effect, unrecognized aspects cannot be ignored in the current compilation; they violate the restriction. This is true despite the Implementation Permissions of 13.1.1. 6.a/5

{*AI12-0389-1*} No_Unrecognized_Pragmas 6.5/5
There are no pragmas having an unrecognized pragma identifier. This restriction applies only to the current compilation or environment, not the entire partition.

Ramification: When this restriction is in effect, unrecognized pragmas cannot be ignored in the current compilation; they violate the restriction. This is true despite the rules of 2.8. 6.b/5

Legality Rules

{*AI95-00381-01*} {*AI12-0444-1*} The restriction_parameter_argument of a No_Dependence restriction shall be a name; the name shall have the form of a full expanded name of a library unit, but can be a name that has no corresponding unit currently present in the environment. 7/5

Ramification: This name is not resolved. 7.a/2

{*AI05-0241-1*} The restriction_parameter_argument of a No_Specification_of_Aspect restriction shall be an identifier; this is an identifier specific to a pragma (see 2.8) and does not denote any declaration. 7.1/3

Ramification: This restriction_parameter_argument is not resolved as it is an identifier specific to a pragma. As for No_Dependence, there is no check that the aspect identifier is meaningful; it might refer to an implementation-defined aspect on one implementation, but nothing at all on another implementation. 7.b/3

{*AI05-0272-1*} The restriction_parameter_argument of a No_Use_Of_Attribute restriction shall be an identifier or one of the reserved words Access, Delta, Digits, Mod, or Range; this is an identifier specific to a pragma. 7.2/3

Ramification: This restriction_parameter_argument is not resolved as it is an identifier specific to a pragma. There is no check that the attribute identifier refers to a known attribute_designator; it might refer to an implementation-defined attribute on one implementation, but nothing at all on another implementation. 7.c/3

{*AI05-0272-1*} The restriction_parameter_argument of a No_Use_Of_Pragma restriction shall be an identifier or the reserved word Interface; this is an identifier specific to a pragma. 7.3/3

Ramification: This restriction_parameter_argument is not resolved as it is an identifier specific to a pragma. There is no check that the pragma identifier refers to a known pragma; it might refer to an implementation-defined pragma on one implementation, but nothing at all on another implementation. 7.d/3

Post-Compilation Rules

8/3 {*AI95-00381-01*} {*AI05-0241-1*} No compilation unit included in the partition shall depend semantically on the library unit identified by the name of a No_Dependence restriction.

8.a/5 **Ramification:** {*AI12-0005-1*} There is no requirement that the library unit actually exists. One possible use of the pragma is to prevent the use of implementation-defined units; when the program is ported to a different compiler, it is perfectly reasonable that no unit with the name exists.

Static Semantics

9/3 {*AI05-0246-1*} The following *profile*_identifier is language defined:

10/3 {*AI05-0246-1*} No_Implementation_Extensions

11/3 {*AI05-0246-1*} For usage profile No_Implementation_Extensions, there shall be no *profile*_pragma_argument_associations.

12/3 {*AI05-0246-1*} The No_Implementation_Extensions usage profile is equivalent to the following restrictions:

13/3
```
No_Implementation_Aspect_Specifications,
No_Implementation_Attributes,
No_Implementation_Identifiers,
No_Implementation_Pragmas,
No_Implementation_Units.
```

Extensions to Ada 95

13.a/2 {*AI95-00257-01*} {*AI95-00368-01*} Restrictions No_Implementation_Attributes, No_Implementation_Pragmas, and No_Obsolescent_Features are new.

13.b/2 {*AI95-00381-01*} Restriction No_Dependence is new.

Extensions to Ada 2005

13.c/3 {*AI05-0241-1*} {*AI05-0242-1*} {*AI05-0246-1*} {*AI05-0272-1*} Restrictions No_Implementation_Aspect_Specifications, No_Implementation_Identifiers, No_Implementation_Units, No_Specification_of_Aspect, No_Use_of_Attribute, and No_Use_of_Pragma are new.

13.d/3 {*AI05-0246-1*} Profile No_Implementation_Extensions is new.

Extensions to Ada 2012

13.e/5 {*AI12-0389-1*} Restrictions No_Unrecognized_Aspects and No_Unrecognized_Pragmas are new.

13.13 Streams

1 A *stream* is a sequence of elements comprising values from possibly different types and allowing sequential access to these values. A *stream type* is a type in the class whose root type is Streams.Root_Stream_Type. A stream type may be implemented in various ways, such as an external sequential file, an internal buffer, or a network channel.

1.a **Discussion:** A stream element will often be the same size as a storage element, but that is not required.

1.b/5 **Term entry: stream** — sequence of elements that can be used, along with the stream-oriented attributes, to support marshalling and unmarshalling of values of most types

Extensions to Ada 83

1.c Streams are new in Ada 95.

13.13.1 The Streams Subsystem

Static Semantics

The abstract type Root_Stream_Type is the root type of the class of stream types. The types in this class represent different kinds of streams. A new stream type is defined by extending the root type (or some other stream type), overriding the Read and Write operations, and optionally defining additional primitive subprograms, according to the requirements of the particular kind of stream. The predefined stream-oriented attributes like T'Read and T'Write make dispatching calls on the Read and Write procedures of the Root_Stream_Type. (User-defined T'Read and T'Write attributes can also make such calls, or can call the Read and Write attributes of other types.) 1

{*AI12-0293-1*} The library package Ada.Streams has the following declaration: 1.1/5

```
{AI12-0241-1} package Ada.Streams                                                    2/5
    with Pure, Nonblocking => False is

{AI95-00161-01}     type Root_Stream_Type is abstract tagged limited private         3/2
        with Preelaborable_Initialization;

{8652/0044} {AI95-00181-01}     type Stream_Element is mod implementation-defined;   4/1
    type Stream_Element_Offset is range implementation-defined;
    subtype Stream_Element_Count is
        Stream_Element_Offset range 0..Stream_Element_Offset'Last;
    type Stream_Element_Array is
        array(Stream_Element_Offset range <>) of aliased Stream_Element;

    procedure Read(                                                                  5
      Stream : in out Root_Stream_Type;
      Item   : out Stream_Element_Array;
      Last   : out Stream_Element_Offset) is abstract;

    procedure Write(                                                                 6
      Stream : in out Root_Stream_Type;
      Item   : in Stream_Element_Array) is abstract;

private                                                                              7
   ... -- not specified by the language
end Ada.Streams;
```

Reason: {*AI12-0241-1*} This package must allow blocking (Nonblocking => False) for compatibility. The purpose of this package is to provide a template for overriding user-defined routines; and such routines can only allow blocking if the root type does so. Users can still declare their overridding routines nonblocking if they wish. 7.a/5

{*AI95-00227-01*} The Read operation transfers stream elements from the specified stream to fill the array Item. Elements are transferred until Item'Length elements have been transferred, or until the end of the stream is reached. If any elements are transferred, the index of the last stream element transferred is returned in Last. Otherwise, Item'First - 1 is returned in Last. Last is less than Item'Last only if the end of the stream is reached. 8/2

The Write operation appends Item to the specified stream. 9

Discussion: {*AI95-00114-01*} The index subtype of Stream_Element_Array is Stream_Element_Offset because we wish to allow maximum flexibility. Most Stream_Element_Arrays will probably have a lower bound of 0 or 1, but other lower bounds, including negative ones, make sense in some situations. 9.a/2

{*AI95-00114-01*} {*AI05-0005-1*} Note that there are some language-defined subprograms that fill part of a Stream_Element_Array, and return the index of the last element filled as a Stream_Element_Offset. The Read procedures declared here, Streams.Stream_IO (see A.12.1), and System.RPC (see E.5) behave in this manner. These will raise Constraint_Error if the resulting Last value is not in Stream_Element_Offset. This implies that the Stream_Element_Array passed to these subprograms should not have a lower bound of Stream_Element_Offset'First, because then a read of 0 elements would always raise Constraint_Error. A better choice of lower bound is 0 or 1. 9.b/3

{*AI12-0293-1*} Three additional packages provide stream implementations that do not make use of any file operations. These packages provide the same operations, with Streams.Storage providing an abstract 10/5

interface, and two child packages providing implementations of that interface. The difference is that for Streams.Storage.Bounded, the maximum storage is bounded.

11/5 {*AI12-0293-1*} The library package Ada.Streams.Storage has the following declaration:

12/5
```
package Ada.Streams.Storage
   with Pure, Nonblocking is

13/5  type Storage_Stream_Type is abstract new Root_Stream_Type with private;

14/5  function Element_Count (Stream : Storage_Stream_Type)
         return Stream_Element_Count is abstract;

15/5  procedure Clear (Stream : in out Storage_Stream_Type) is abstract;

16/5 private
   ... -- not specified by the language
end Ada.Streams.Storage;
```

17/5 {*AI12-0293-1*} The library package Ada.Streams.Storage.Unbounded has the following declaration:

18/5
```
package Ada.Streams.Storage.Unbounded
   with Prelaborated, Nonblocking, Global => in out synchronized is

19/5  type Stream_Type is new Storage_Stream_Type with private
         with Default_Initial_Condition =>
             Element_Count (Stream_Type) = 0;

20/5  overriding
      procedure Read (
         Stream : in out Stream_Type;
         Item   : out Stream_Element_Array;
         Last   : out Stream_Element_Offset)
         with Post =>
             (declare
                 Num_Read : constant Stream_Element_Count :=
                    Stream_Element_Count'Min
                       (Element_Count(Stream)'Old, Item'Length);
              begin
                 Last = Num_Read + Item'First - 1 and
                 Element_Count (Stream) =
                    Element_Count (Stream)'Old - Num_Read);
```

20.a/5 **Discussion:** Num_Read is the number of elements read; this is the minimum of Item'Length and the number of available elements. Last then is determined by that, and the Element_Count decreases by the number of elements read.

21/5
```
      overriding
      procedure Write (
         Stream : in out Stream_Type;
         Item   : in Stream_Element_Array)
         with Post =>
            Element_Count (Stream) =
            Element_Count (Stream)'Old + Item'Length;

22/5  overriding
      function Element_Count (Stream : Stream_Type)
         return Stream_Element_Count;

23/5  overriding
      procedure Clear (Stream : in out Stream_Type)
         with Post => Element_Count (Stream) = 0;

24/5 private
   ... -- not specified by the language
end Ada.Streams.Storage.Unbounded;
```

25/5 {*AI12-0293-1*} The library package Ada.Streams.Storage.Bounded has the following declaration:

26/5
```
package Ada.Streams.Storage.Bounded
    with Pure, Nonblocking is

27/5  type Stream_Type (Max_Elements : Stream_Element_Count)
         is new Storage_Stream_Type with private
         with Default_Initial_Condition =>
            Element_Count (Stream_Type) = 0;
```

```
   overriding                                                                    28/5
   procedure Read (
      Stream : in out Stream_Type;
      Item   : out Stream_Element_Array;
      Last   : out Stream_Element_Offset)
      with Post =>
          (declare
              Num_Read : constant Stream_Element_Count :=
                 Stream_Element_Count'Min
                    (Element_Count(Stream)'Old, Item'Length);
           begin
              Last = Num_Read + Item'First - 1 and
              Element_Count (Stream) =
                 Element_Count (Stream)'Old - Num_Read);

   overriding                                                                    29/5
   procedure Write (
      Stream : in out Stream_Type;
      Item   : in Stream_Element_Array)
      with Pre =>
              Element_Count (Stream) + Item'Length <= Stream.Max_Elements
              or else (raise Constraint_Error),
           Post =>
              Element_Count (Stream) =
              Element_Count (Stream)'Old + Item'Length;

   overriding                                                                    30/5
   function Element_Count (Stream : Stream_Type)
      return Stream_Element_Count
      with Post => Element_Count'Result <= Stream.Max_Elements;

   overriding                                                                    31/5
    procedure Clear (Stream : in out Stream_Type)
       with Post => Element_Count (Stream) = 0;

private                                                                          32/5
   ... -- not specified by the language
end Ada.Streams.Storage.Bounded;
```

{*AI12-0293-1*} The Element_Count functions return the number of stream elements that are available for reading from the given stream. 33/5

{*AI12-0293-1*} The Read and Write procedures behave as described for package Ada.Streams above. Stream elements are read in FIFO (first-in, first-out) order; stream elements are available for reading immediately after they are written. 34/5

{*AI12-0293-1*} The Clear procedures remove any available stream elements from the given stream. 35/5

Implementation Permissions

{*8652/0044*} {*AI95-00181-01*} {*AI12-0444-1*} If Stream_Element'Size is not a multiple of System.Storage_Unit, then the components of Stream_Element_Array will not be aliased. 36/5

Ramification: {*AI95-00114-01*} If the Stream_Element'Size is less than the size of System.Storage_Unit, then components of Stream_Element_Array need not be aliased. This is necessary as the components of type Stream_Element size might not be addressable on the target architecture. 36.a/2

Implementation Advice

{*AI12-0293-1*} Streams.Storage.Bounded.Stream_Type objects should be implemented without implicit pointers or dynamic allocation. 37/5

Implementation Advice: Streams.Storage.Bounded.Stream_Type objects should be implemented without implicit pointers or dynamic allocation. 37.a.1/5

Reason: The Streams.Storage.Bounded package is provided in order to make available an alternative to the Streams.Storage.Unbounded package which gives more predictable memory usage. 37.a/5

NOTE 1 See A.12.1, “The Package Streams.Stream_IO” for an example of extending type Root_Stream_Type. 38

39/2 NOTE 2 {*AI95-00227-01*} If the end of stream has been reached, and Item'First is Stream_Element_Offset'First, Read will raise Constraint_Error.

39.a/2 **Ramification:** Thus, Stream_Element_Arrays should start at 0 or 1, not Stream_Element_Offset'First.

Extensions to Ada 95

39.b/2 {*AI95-00161-01*} **Amendment Correction:** Added pragma Preelaborable_Initialization to type Root_Stream_Type.

Wording Changes from Ada 95

39.c/2 {*8652/0044*} {*AI95-00181-01*} **Corrigendum:** Stream elements are aliased presuming that makes sense.

39.d/2 {*AI95-00227-01*} Fixed the wording for Read to properly define the result in Last when no stream elements are transfered.

Extensions to Ada 2012

39.e/5 {*AI12-0293-1*} Package Ada.Streams.Storage and its children are new.

13.13.2 Stream-Oriented Attributes

1/3 {*8652/0009*} {*AI95-00137-01*} {*AI05-0183-1*} The type-related operational attributes Write, Read, Output, and Input convert values to a stream of elements and reconstruct values from a stream.

Static Semantics

1.1/2 {*AI95-00270-01*} For every subtype S of an elementary type *T*, the following representation attribute is defined:

1.2/3 S'Stream_Size

{*AI95-00270-01*} {*AI05-0194-1*} Denotes the number of bits read from or written to a stream by the default implementations of S'Read and S'Write. Hence, the number of stream elements required per item of elementary type *T* is:

1.3/2
```
T'Stream_Size / Ada.Streams.Stream_Element'Size
```

1.4/2 The value of this attribute is of type *universal_integer* and is a multiple of Stream_Element'Size.

1.5/2 Stream_Size may be specified for first subtypes via an attribute_definition_clause; the expression of such a clause shall be static, nonnegative, and a multiple of Stream_Element'Size.

1.a/3 **Aspect Description for Stream_Size:** Size in bits used to represent elementary objects in a stream.

1.b/2 **Discussion:** Stream_Size is a type-related attribute (see 13.1).

1.c/3 **Ramification:** {*AI05-0194-1*} The value of S'Stream_Size is unaffected by the presence or absence of any attribute_definition_clauses or aspect_specifications specifying the Read or Write attributes of any ancestor of S. S'Stream_Size is defined in terms of the behavior of the default implementations of S'Read and S'Write even if those default implementations are overridden.

Implementation Advice

1.6/2 {*AI95-00270-01*} If not specified, the value of Stream_Size for an elementary type should be the number of bits that corresponds to the minimum number of stream elements required by the first subtype of the type, rounded up to the nearest factor or multiple of the word size that is also a multiple of the stream element size.

1.d/2 **Implementation Advice:** If not specified, the value of Stream_Size for an elementary type should be the number of bits that corresponds to the minimum number of stream elements required by the first subtype of the type, rounded up to the nearest factor or multiple of the word size that is also a multiple of the stream element size.

1.e/2 **Reason:** {*AI95-00270-01*} This is Implementation Advice because we want to allow implementations to remain compatible with their Ada 95 implementations, which may have a different handling of the number of stream elements. Users can always specify Stream_Size if they need a specific number of stream elements.

{*AI95-00270-01*} The recommended level of support for the Stream_Size attribute is: 1.7/2

- {*AI95-00270-01*} A Stream_Size clause should be supported for a discrete or fixed point type *T* if the specified Stream_Size is a multiple of Stream_Element'Size and is no less than the size of the first subtype of *T*, and no greater than the size of the largest type of the same elementary class (signed integer, modular integer, enumeration, ordinary fixed point, or decimal fixed point). 1.8/2

  Implementation Advice: The recommended level of support for the Stream_Size attribute should be followed. 1.f/2

  Ramification: There are no requirements beyond supporting confirming Stream_Size clauses for floating point and access types. Floating point and access types usually only have a handful of defined formats, streaming anything else makes no sense for them. 1.g/2

  For discrete and fixed point types, this may require support for sizes other than the “natural” ones. For instance, on a typical machine with 32-bit integers and a Stream_Element'Size of 8, setting Stream_Size to 24 must be supported. This is required as such formats can be useful for interoperability with unusual machines, and there is no difficulty with the implementation (drop extra bits on output, sign extend on input). 1.h/2

Static Semantics

For every subtype S of a specific type *T*, the following attributes are defined. 2

S'Write S'Write denotes a procedure with the following specification: 3

```
{AI95-00441-01} procedure S'Write(
   Stream : not null access Ada.Streams.Root_Stream_Type'Class;
   Item : in T)
```
4/2

S'Write writes the value of *Item* to *Stream*. 5

S'Read S'Read denotes a procedure with the following specification: 6

```
{AI95-00441-01} procedure S'Read(
   Stream : not null access Ada.Streams.Root_Stream_Type'Class;
   Item : out T)
```
7/2

S'Read reads the value of *Item* from *Stream*. 8

This paragraph was deleted.{*8652/0040*} {*AI95-00108-01*} {*AI95-00444-01*} {*AI05-0192-1*} {*AI12-0419-1*} 8.1/5

{*AI95-00444-01*} The default implementations of the Write and Read attributes, where available, execute as follows: 8.2/2

{*8652/0040*} {*AI95-00108-01*} {*AI95-00195-01*} {*AI95-00251-01*} {*AI95-00270-01*} {*AI05-0139-2*} {*AI12-0191-1*} {*AI12-0419-1*} For nonderived elementary types, Read reads (and Write writes) the number of stream elements implied by the Stream_Size for the type *T*; the representation of those stream elements is implementation defined. For nonderived composite types, the Write or Read attribute for each component (excluding those, if any, that are not components of the nominal type of the object) is called in canonical order, which is last dimension varying fastest for an array (unless the convention of the array is Fortran, in which case it is first dimension varying fastest), and positional aggregate order for a record. Bounds are not included in the stream if *T* is an array type. If *T* is a discriminated type, discriminants are included only if they have defaults. If *T* is a tagged type, the tag is not included. 9/5

Implementation defined: The contents of the stream elements read and written by the Read and Write attributes of elementary types. 9.a/2

{*AI12-0419-1*} For type extensions, the Write or Read attribute for the parent type is called, followed by the Write or Read attribute of each component of the extension part, in canonical order. For a limited type extension, if the attribute of the parent type or any progenitor type of *T* is available anywhere within the immediate scope of *T*, and the attribute of the parent type or the type of any of the extension components is not available at the freezing point of *T*, then the attribute of *T* shall be directly specified. For untagged derived types, the Write (resp. Read) attribute invokes the corresponding attribute of the parent type, if the attribute is available for the parent type. 9.1/5

9.a.1/5 **Implementation Note:** {*AI12-0419-1*} The default implementation of a stream-oriented attribute might not be well-defined in cases where the attribute is never available anywhere in the scope of the type. In such cases, the attribute cannot be called and is defined (far below) to raise Program_Error.

9.2/5 {*AI05-0023-1*} {*AI05-0264-1*} {*AI12-0419-1*} If *T* is a discriminated type and its discriminants have defaults, then S'Read first reads the discriminants from the stream without modifying *Item*. S'Read then creates an object of type *T* constrained by these discriminants. The value of this object is then converted to the subtype of *Item* and is assigned to *Item*. Finally, the Read attribute for each nondiscriminant component of *Item* is called in canonical order as described above. Normal default initialization and finalization take place for the created object.

9.b **Reason:** A discriminant with a default value is treated simply as a component of the object. On the other hand, an array bound or a discriminant without a default value, is treated as "descriptor" or "dope" that must be provided in order to create the object and thus is logically separate from the regular components. Such "descriptor" data are written by 'Output and produced as part of the delivered result by the 'Input function, but they are not written by 'Write nor read by 'Read. A tag is like a discriminant without a default.

9.b.1/1 {*8652/0040*} {*AI95-00108-01*} For limited type extensions, we must have a definition of 'Read and 'Write if the parent type has one, as it is possible to make a dispatching call through the attributes. The rule is designed to automatically do the right thing in as many cases as possible.

9.b.2/1 {*AI95-00251-01*} Similarly, a type that has a progenitor with an available attribute must also have that attribute, for the same reason.

9.b.3/3 {*AI05-0023-1*} The semantics of S'Read for a discriminated type with defaults involves an anonymous object so that the point of required initialization and finalization is well-defined, especially for objects that change shape and have controlled components. The creation of this anonymous object often can be omitted (see the Implementation Permissions below).

9.c/2 **Ramification:** {*AI95-00195-01*} For a composite object, the subprogram denoted by the Write or Read attribute of each component is called, whether it is the default or is user-specified. Implementations are allowed to optimize these calls (see below), presuming the properties of the attributes are preserved.

9.3/5 {*AI95-00270-01*} {*AI05-0264-1*} {*AI12-0419-1*} Constraint_Error is raised by the predefined Write attribute if the value of the elementary item is outside the range of values representable using Stream_Size bits. For a signed integer type, an enumeration type, or a fixed point type, the range is unsigned only if the integer code for the lower bound of the first subtype is nonnegative, and a (symmetric) signed range that covers all values of the first subtype would require more than Stream_Size bits; otherwise, the range is signed.

10 For every subtype S'Class of a class-wide type *T*'Class:

11 S'Class'Write

S'Class'Write denotes a procedure with the following specification:

12/2
```
{AI95-00441-01} procedure S'Class'Write(
   Stream : not null access Ada.Streams.Root_Stream_Type'Class;
   Item   : in T'Class)
```

13 Dispatches to the subprogram denoted by the Write attribute of the specific type identified by the tag of Item.

14 S'Class'Read

S'Class'Read denotes a procedure with the following specification:

15/2
```
{AI95-00441-01} procedure S'Class'Read(
   Stream : not null access Ada.Streams.Root_Stream_Type'Class;
   Item : out T'Class)
```

16 Dispatches to the subprogram denoted by the Read attribute of the specific type identified by the tag of Item.

16.a **Reason:** It is necessary to have class-wide versions of Read and Write in order to avoid generic contract model violations; in a generic, we don't necessarily know at compile time whether a given type is specific or class-wide.

Paragraph 17 was deleted.

Static Semantics

For every subtype S of a specific type *T*, the following attributes are defined. 18

S'Output S'Output denotes a procedure with the following specification: 19

```
{AI95-00441-01} procedure S'Output(                                          20/2
   Stream : not null access Ada.Streams.Root_Stream_Type'Class;
   Item : in T)
```

S'Output writes the value of *Item* to *Stream*, including any bounds or discriminants. 21

Ramification: Note that the bounds are included even for an array type whose first subtype is constrained. 21.a

S'Input S'Input denotes a function with the following specification: 22

```
{AI95-00441-01} function S'Input(                                            23/2
   Stream : not null access Ada.Streams.Root_Stream_Type'Class)
   return T
```

S'Input reads and returns one value from *Stream*, using any bounds or discriminants written by a corresponding S'Output to determine how much to read. 24

{*8652/0040*} {*AI95-00108-01*} {*AI95-00444-01*} {*AI05-0192-1*} {*AI12-0419-1*} For an untagged derived type, the default implementation of the Output (resp. Input) attribute invokes the corresponding attribute of the parent type, if the attribute is available for the parent type. For any other type, the default implementations of the Output and Input attributes, where available, execute as follows: 25/5

- {*AI05-0269-1*} If *T* is an array type, S'Output first writes the bounds, and S'Input first reads the bounds. If *T* has discriminants without defaults, S'Output first writes the discriminants (using the Write attribute of the discriminant type for each), and S'Input first reads the discriminants (using the Read attribute of the discriminant type for each). 26/3
- {*AI95-00195-01*} {*AI05-0023-1*} S'Output then calls S'Write to write the value of *Item* to the stream. S'Input then creates an object of type *T*, with the bounds or (when without defaults) the discriminants, if any, taken from the stream, passes it to S'Read, and returns the value of the object. If *T* has discriminants, then this object is unconstrained if and only the discriminants have defaults. Normal default initialization and finalization take place for this object (see 3.3.1, 7.6, and 7.6.1). 27/3

{*AI95-00251-01*} If *T* is an abstract type, then S'Input is an abstract function. 27.1/2

Ramification: For an abstract type *T*, S'Input can be called in a dispatching call, or passed to an abstract formal subprogram. But it cannot be used in nondispatching contexts, because we don't allow objects of abstract types to exist. The designation of this function as abstract has no impact on descendants of *T*, as *T*'Input is not inherited for tagged types, but rather recreated (and the default implementation of *T*'Input calls *T*'Read, not the parent type's *T*'Input). Note that *T*'Input cannot be specified in this case, as any function with the proper profile is necessarily abstract, and specifying abstract subprograms in an attribute_definition_clause is illegal. 27.a/2

For every subtype S'Class of a class-wide type *T*'Class: 28

S'Class'Output 29

S'Class'Output denotes a procedure with the following specification:

```
{AI95-00441-01} procedure S'Class'Output(                                    30/2
   Stream : not null access Ada.Streams.Root_Stream_Type'Class;
   Item   : in T'Class)
```

{*AI95-00344-01*} First writes the external tag of *Item* to *Stream* (by calling String'Output(*Stream*, Tags.External_Tag(*Item*'Tag)) — see 3.9) and then dispatches to the subprogram denoted by the Output attribute of the specific type identified by the tag. Tag_Error is raised if the tag of Item identifies a type declared at an accessibility level deeper than that of S. 31/2

Reason: {*AI95-00344-01*} We raise Tag_Error here for nested types as such a type cannot be successfully read with S'Class'Input, and it doesn't make sense to allow writing a value that cannot be read. 31.a/2

32 S'Class'Input

S'Class'Input denotes a function with the following specification:

33/2
```
{AI95-00441-01} function S'Class'Input(
   Stream : not null access Ada.Streams.Root_Stream_Type'Class)
   return T'Class
```

34/5 {*AI95-00279-01*} {*AI95-00344-01*} {*AI05-0109-1*} {*AI12-0439-1*} First reads the external tag from *Stream* and determines the corresponding internal tag (by calling Tags.Descendant_Tag(String'Input(*Stream*), S'Tag) which can raise Tag_Error — see 3.9) and then dispatches to the subprogram denoted by the Input attribute of the specific type identified by the internal tag; returns that result. If the specific type identified by the internal tag is abstract, Constraint_Error is raised.

34.a/3 **Ramification:** {*AI05-0109-1*} Descendant_Tag will ensure that the tag it returns is covered by T'Class; Tag_Error will be raised if it would not cover T'Class.

35/3 {*AI95-00195-01*} {*AI05-0228-1*} In the default implementation of Read and Input for a composite type, for each scalar component that is a discriminant or that has an implicit initial value, a check is made that the value returned by Read for the component belongs to its subtype. Constraint_Error is raised if this check fails. For other scalar components, no check is made. For each component that is of an access type, if the implementation can detect that the value returned by Read for the component is not a value of its subtype, Constraint_Error is raised. If the value is not a value of its subtype and this error is not detected, the component has an abnormal value, and erroneous execution can result (see 13.9.1). In the default implementation of Read for a composite type with defaulted discriminants, if the actual parameter of Read is constrained, a check is made that the discriminants read from the stream are equal to those of the actual parameter. Constraint_Error is raised if this check fails.

35.a/3 **Reason:** {*AI05-0228-1*} The check for scalar components that have an implicit initial value is to preserve our Language Design Principle that all objects that have an implicit initial value do not become "deinitialized".

35.b/3 **Ramification:** {*AI05-0228-1*} A scalar component can have an implicit initial value if it has a default_expression, if the component's type has the Default_Value aspect specified, or if the component is that of an array type that has the Default_Component_Value aspect specified.

35.c/3 **To be honest:** {*AI05-0228-1*} An implementation should always be able to detect the error for a null value read into a component of an access subtype with a null exclusion; the "if the implementation can detect" is intended to cover nonnull access values.

36/2 {*AI95-00195-01*} It is unspecified at which point and in which order these checks are performed. In particular, if Constraint_Error is raised due to the failure of one of these checks, it is unspecified how many stream elements have been read from the stream.

37/1 {*8652/0045*} {*AI95-00132-01*} In the default implementation of Read and Input for a type, End_Error is raised if the end of the stream is reached before the reading of a value of the type is completed.

37.1/5 {*AI12-0064-2*} {*AI12-0396-1*} The Nonblocking aspect is statically True and the Global aspect is **null** for the default implementations of stream-oriented attributes for elementary types. For the default implementations of stream-oriented attributes for composite types, the value of the Nonblocking aspect is that of the first subtype, and the Global aspect defaults to that of the first subtype. A default implementation of a stream-oriented attribute that has the Nonblocking aspect statically True is considered a nonblocking region. The aspect Dispatching (see H.7.1) is Read(Stream) for the default implementations of the stream-oriented attributes Read, Read'Class, Input, and Input'Class; the aspect Dispatching is Write(Stream) for the default implementations of the stream-oriented attributes Write, Write'Class, Output, and Output'Class.

37.a/5 **Ramification:** If the default implementation of a stream-oriented attribute *A* for a composite type would call other stream-oriented attribute(s) whose Global aspect was not allowed by the Global aspect of*A*, then the Legality Rules of 6.1.2 are violated and the type declaration is illegal. If the stream-oriented attribute *A* is overridden, then the default implementation is not created and this check is not made. A similar ramification applies to the Nonblocking aspect of default implementation of a stream-oriented attribute.

{*8652/0040*} {*AI95-00108-01*} {*AI95-00195-01*} {*AI95-00251-01*} {*AI05-0039-1*} {*AI12-0106-1*} {*AI12-0121-1*} {*AI12-0435-1*} The stream-oriented attributes may be specified for any type via an attribute_definition_clause. [Alternatively, each of the specific stream-oriented attributes may be specified using an aspect_specification on any type_declaration, with the aspect name being the corresponding attribute name.] Each of the class-wide stream-oriented attributes may be specified using an aspect_specification for a tagged type *T* using the name of the stream-oriented attribute followed by 'Class; such class-wide aspects do not apply to other descendants of *T*. If not directly specified, a default implementation of a stream-oriented attribute is implicitly composed for a nonlimited type, and for certain limited types, as defined above. 38/5

Reason: {*AI95-00195-01*} {*AI12-0106-1*} We need the last sentence to override the blanket rule given in 13.1.1 that aspect'Class applies to the type and all descendants. 38.a/4

This paragraph was deleted.{*8652/0040*} {*AI95-00108-01*} {*AI95-00195-01*} 38.a.1/2

Proof: {*AI12-0121-1*} 13.1.1 says that all operational attributes can be specified with an aspect_specification. 38.a.2/4

Aspect Description for Read'Class: Procedure to read a value from a stream for the class-wide type associated with a given type. 38.a.3/4

Aspect Description for Write'Class: Procedure to write a value to a stream for a the class-wide type associated with a given type. 38.a.4/4

Aspect Description for Input'Class: Function to read a value from a stream for a the class-wide type associated with a given type, including any bounds and discriminants. 38.a.5/4

Aspect Description for Output'Class: Procedure to write a value to a stream for a the class-wide type associated with a given type, including any bounds and discriminants. 38.a.6/4

{*AI12-0121-1*} The subprogram name given in such an attribute_definition_clause or aspect_specification shall statically denote a subprogram that is not an abstract subprogram. Furthermore, if a specific stream-oriented attribute is specified for an interface type, the subprogram name given in the attribute_definition_clause or aspect_specification shall statically denote a null procedure. 38.1/4

Discussion: {*AI95-00251-01*} Stream attributes (other than Input) are always null procedures for interface types (they have no components). We need to allow explicit setting of the Read and Write attributes in order that the class-wide attributes like LI'Class'Input can be made available. (In that case, any descendant of the interface type would require available attributes.) But we don't allow any concrete implementation because these don't participate in extensions (unless the interface is the parent type). If we didn't ban concrete implementations, the order of declaration of a pair of interfaces would become significant. For example, if Int1 and Int2 are interfaces with concrete implementations of 'Read, then the following declarations would have different implementations for 'Read: 38.b/2

```
type Con1 is new Int1 and Int2 with null record;
type Con2 is new Int2 and Int1 with null record;
```
38.c/2

This would violate our design principle that the order of the specification of the interfaces in a derived_type_definition doesn't matter. 38.d/2

Ramification: The Input attribute cannot be specified for an interface. As it is a function, a null procedure is impossible; a concrete function is not possible anyway as any function returning an abstract type must be abstract. And we don't allow specifying stream attributes to be abstract subprograms. This has no impact, as the availability of Int'Class'Input (where Int is a limited interface) depends on whether Int'Read (not Int'Input) is specified. There is no reason to allow Int'Output to be specified, either, but there is equally no reason to disallow it, so we don't have a special rule for that. 38.e/2

Discussion: {*AI95-00195-01*} Limited types generally do not have default implementations of the stream-oriented attributes. The rules defining when a stream-oriented attribute is available (see below) determine when an attribute of a limited type is in fact well defined and usable. The rules are designed to maximize the number of cases in which the attributes are usable. For instance, when the language provides a default implementation of an attribute for a limited type based on a specified attribute for the parent type, we want to be able to call that attribute. 38.f/2

Aspect Description for Read: Procedure to read a value from a stream for a given type. 38.g/3

Aspect Description for Write: Procedure to write a value to a stream for a given type. 38.h/3

Aspect Description for Input: Function to read a value from a stream for a given type, including any bounds and discriminants. 38.i/3

Aspect Description for Output: Procedure to write a value to a stream for a given type, including any bounds and discriminants. 38.j/3

39/2 {*AI95-00195-01*} A stream-oriented attribute for a subtype of a specific type *T* is *available* at places where one of the following conditions is true:

40/2 • *T* is nonlimited.

41/2 • The attribute_designator is Read (resp. Write) and *T* is a limited record extension, and the attribute Read (resp. Write) is available for the parent type of *T* and for the types of all of the extension components.

41.a/2 **Reason:** In this case, the language provides a well-defined default implementation, which we want to be able to call.

42/5 • {*AI12-0419-1*} *T* is a limited untagged derived type, and the attribute is available for the parent type.

42.a/5 **Reason:** {*AI12-0419-1*} Again, the language provides a default implementation, which we want to be able to call.

43/2 • The attribute_designator is Input (resp. Output), and *T* is a limited type, and the attribute Read (resp. Write) is available for *T*.

43.a/2 **Reason:** The default implementation of Input and Output are based on Read and Write; so if the implementation of Read or Write is good, so is the matching implementation of Input or Output.

44/5 • {*AI12-0419-1*} The attribute has been specified via an attribute_definition_clause or aspect_specification, and the attribute_definition_clause or aspect_specification is visible.

44.a/2 **Reason:** We always want to allow calling a specified attribute. But we don't want availability to break privacy. Therefore, only attributes whose specification can be seen count. Yes, we defined the visibility of an attribute_definition_clause (see 8.3).

45/2 {*AI95-00195-01*} A stream-oriented attribute for a subtype of a class-wide type *T*'Class is available at places where one of the following conditions is true:

46/2 • *T* is nonlimited;

47/5 • {*AI12-0419-1*} the attribute has been specified via an attribute_definition_clause or aspect_specification, and the attribute_definition_clause or aspect_specification is visible; or

48/2 • the corresponding attribute of *T* is available, provided that if *T* has a partial view, the corresponding attribute is available at the end of the visible part where *T* is declared.

48.a/2 **Reason:** The rules are stricter for class-wide attributes because (for the default implementation) we must ensure that any specific attribute that might ever be dispatched to is available. Because we require specification of attributes for extensions of limited parent types with available attributes, we can in fact know this. Otherwise, we would not be able to use default class-wide attributes with limited types, a significant limitation.

49/4 {*AI95-00195-01*} {*AI12-0030-1*} An attribute_reference for one of the stream-oriented attributes is illegal unless the attribute is available at the place of the attribute_reference. Furthermore, an attribute_reference for *T*'Input is illegal if *T* is an abstract type. In addition to the places where Legality Rules normally apply (see 12.3), these rules also apply in the private part of an instance of a generic unit.

49.a/2 **Discussion:** Stream attributes always exist. It is illegal to call them in some cases. Having the attributes not be defined for some limited types would seem to be a cleaner solution, but it would lead to contract model problems for limited private types.

49.b/2 *T*'Input is available for abstract types so that *T*'Class'Input is available. But we certainly don't want to allow calls that could create an object of an abstract type. Remember that *T*'Class is never abstract, so the above legality rule doesn't apply to it. We don't have to discuss whether the attribute is specified, as it cannot be: any function returning the type would have to be abstract, and we do not allow specifying an attribute with an abstract subprogram.

49.c/4 **To be honest:** {*AI12-0030-1*} "These rules apply" refers to just this paragraph and not to the rest of the rules in this section. This rule probably should have been a Legality Rule, but the word "illegal" should key the reader that this is a Legality Rule, no matter under what text heading it occurs.

49.1/5 {*AI12-0030-1*} {*AI12-0419-1*} Unless available for a parent type, if any, for an untagged type having a task, protected, or explicitly limited record part, the default implementation of each of the Read, Write, Input, and Output attributes raises Program_Error and performs no other action.

Discussion: {*AI12-0030-1*} {*AI12-0419-1*} It might seem that there is no need to specify the behavior of the default implementation of a streaming attribute of, for example, a task type because there is no way that it can be invoked. It is possible, however, to construct an example where such a stream attribute can be invoked. This involves using a formal untagged limited derived type for which some streaming attribute is available (because it was explicitly specified for the ancestor type) and a corresponding actual type for which the attribute is unspecified (because the derivation occurred in a place where the attribute is never available for the parent type). 49.d/5

{*AI95-00195-01*} {*AI05-0192-1*} In the parameter_and_result_profiles for the default implementations of the stream-oriented attributes, the subtype of the *Item* parameter is the base subtype of *T* if *T* is a scalar type, and the first subtype otherwise. The same rule applies to the result of the Input attribute. 50/3

{*AI95-00195-01*} {*AI05-0007-1*} {*AI12-0427-1*} For an attribute_definition_clause or aspect_specification specifying one of these attributes, the subtype of the *Item* parameter shall be the first subtype or the base subtype if scalar, and the first subtype if not scalar. The same rule applies to the result of the Input function. 51/5

Reason: This is to simplify implementation. 51.a/2

Ramification: {*AI12-0427-1*} The view of the type at the point of the attribute_definition_clause determines whether the base subtype is allowed. Thus, for a scalar type with a partial view (which is never scalar), whether the base subtype is allowed is determined by whether the attribute_definition_clause occurs before or after the full definition of the scalar type. For the same reason, the base subtype is never allowed for an attribute specified via an aspect_specification on the partial view. 51.b/5

{*AI95-00366-01*} {*AI05-0065-1*} [A type is said to *support external streaming* if Read and Write attributes are provided for sending values of such a type between active partitions, with Write marshalling the representation, and Read unmarshalling the representation.] A limited type supports external streaming only if it has available Read and Write attributes. A type with a part that is of a nonremote access type supports external streaming only if that access type or the type of some part that includes the access type component, has Read and Write attributes that have been specified via an attribute_definition_clause, and that attribute_definition_clause is visible. [An anonymous access type does not support external streaming.]All other types (including remote access types, see E.2.2) support external streaming. 52/3

Ramification: A limited type with a part that is of a nonremote access type needs to satisfy both rules. 52.a/3

Erroneous Execution

{*AI95-00279-01*} {*AI95-00344-01*} If the internal tag returned by Descendant_Tag to T'Class'Input identifies a type that is not library-level and whose tag has not been created, or does not exist in the partition at the time of the call, execution is erroneous. 53/2

Ramification: The definition of Descendant_Tag prevents such a tag from being provided to T'Class'Input if T is a library-level type. However, this rule is needed for nested tagged types. 53.a/2

Implementation Requirements

{*8652/0040*} {*AI95-00108-01*} For every subtype *S* of a language-defined nonlimited specific type *T*, the output generated by S'Output or S'Write shall be readable by S'Input or S'Read, respectively. This rule applies across partitions if the implementation conforms to the Distributed Systems Annex. 54/1

{*AI95-00195-01*} {*AI05-0092-1*} If Constraint_Error is raised during a call to Read because of failure of one the above checks, the implementation shall ensure that the discriminants of the actual parameter of Read are not modified. 55/3

Implementation Permissions

{*AI95-00195-01*} {*AI05-0092-1*} {*AI12-0445-1*} The number of calls performed by the predefined implementation of the stream-oriented attributes on the Read and Write operations of the stream type is unspecified. An implementation may take advantage of this permission to perform internal buffering. 56/5

However, all the calls on the Read and Write operations of the stream type used to implement an explicit invocation of a stream-oriented attribute shall take place before this invocation returns. An explicit invocation is one appearing explicitly in the program text, possibly through a generic instantiation (see 12.3).

56.1/5 {*AI05-0023-1*} {*AI05-0264-1*} {*AI12-0445-1*} If *T* is a discriminated type and its discriminants have defaults, then in two cases an execution of the default implementation of S'Read is not required to create an anonymous object of type *T*: If the discriminant values that are read in are equal to the corresponding discriminant values of *Item*, then creation of a new object of type *T* may be bypassed and *Item* may be used instead. If they are not equal and *Item* is a constrained variable, then Constraint_Error may be raised at that point, before any further values are read from the stream and before the object of type *T* is created.

56.2/3 {*AI05-0023-1*} A default implementation of S'Input that calls the default implementation of S'Read may create a constrained anonymous object with discriminants that match those in the stream.

56.a/3 **Implementation Note:** This allows the combined executions of S'Input and S'Read to create one object of type *T* instead of two. If this option is exercised, then:

56.b/3 - The discriminants are read from the stream by S'Input, not S'Read.

56.c/3 - S'Input declares an object of type *T* constrained by the discriminants read from the stream, not an unconstrained object.

56.d/3 - The discriminant values that S'Read would normally have read from the stream are read from Item instead.

56.e/3 - The permissions of the preceding paragraph then apply and no object of type *T* need be created by the execution of S'Read.

57/5 NOTE 1 {*AI12-0442-1*} For a definite subtype S of a type *T*, only *T*'Write and *T*'Read are necessary to pass an arbitrary value of the subtype through a stream. For an indefinite subtype S of a type *T*, *T*'Output and *T*'Input will normally be necessary, since *T*'Write and *T*'Read do not pass bounds, discriminants, or tags.

58 NOTE 2 User-specified attributes of S'Class are not inherited by other class-wide types descended from S.

Examples

59 *Example of user-defined Write attribute:*

60/2
```
{AI95-00441-01} procedure My_Write(
  Stream : not null access Ada.Streams.Root_Stream_Type'Class;
  Item   : My_Integer'Base);
for My_Integer'Write use My_Write;
```

60.a **Discussion:** *Example of network input/output using input output attributes:*

60.b
```
with Ada.Streams; use Ada.Streams;
generic
    type Msg_Type(<>) is private;
package Network_IO is
    -- Connect/Disconnect are used to establish the stream
    procedure Connect(...);
    procedure Disconnect(...);
```

60.c
```
    -- Send/Receive transfer messages across the network
    procedure Send(X : in Msg_Type);
    function Receive return Msg_Type;
private
    type Network_Stream is new Root_Stream_Type with ...
    procedure Read(...);  -- define Read/Write for Network_Stream
    procedure Write(...);
end Network_IO;
```

60.d
```
with Ada.Streams; use Ada.Streams;
package body Network_IO is
    Current_Stream : aliased Network_Stream;
    . . .
    procedure Connect(...) is ...;
    procedure Disconnect(...) is ...;
```

60.e
```
    procedure Send(X : in Msg_Type) is
    begin
        Msg_Type'Output(Current_Stream'Access, X);
    end Send;
```

```
    function Receive return Msg_Type is                                    60.f
    begin
        return Msg_Type'Input(Current_Stream'Access);
    end Receive;
end Network_IO;
```

Inconsistencies With Ada 95

{*8652/0040*} {*AI95-00108-01*} **Corrigendum:** Clarified how the default implementation for stream attributes is determined (eliminating conflicting language). The new wording provides that attributes for type extensions are created by composing the parent's attribute with those for the extension components if any. If a program was written assuming that the extension components were not included in the stream (as in original Ada 95), it would fail to work in the language as corrected by the Corrigendum. 60.g/2

{*AI95-00195-01*} **Amendment Correction:** Explicitly provided a permission that the number of calls to the underlying stream Read and Write operations may differ from the number determined by the canonical operations. If Ada 95 code somehow depended on the number of calls to Read or Write, it could fail with an Ada 2005 implementation. Such code is likely to be very rare; moreover, such code is really wrong, as the permission applies to Ada 95 as well. 60.h/2

Extensions to Ada 95

{*AI95-00270-01*} The Stream_Size attribute is new. It allows specifying the number of bits that will be streamed for a type. The Implementation Advice involving this also was changed; this is not incompatible because Implementation Advice does not have to be followed. 60.i/2

{*8652/0040*} {*AI95-00108-01*} {*AI95-00195-01*} {*AI95-00444-01*} **Corrigendum:** Limited types may have default constructed attributes if all of the parent and (for extensions) extension components have available attributes. Ada 2005 adds the notion of availability to patch up some holes in the Corrigendum model. 60.j/2

Wording Changes from Ada 95

{*8652/0009*} {*AI95-00137-01*} **Corrigendum:** Added wording to specify that these are operational attributes. 60.k/2

{*8652/0045*} {*AI95-00132-01*} **Corrigendum:** Clarified that End_Error is raised by the default implementation of Read and Input if the end of the stream is reached. (The result could have been abnormal without this clarification, thus this is not an inconsistency, as the programmer could not have depended on the previous behavior.) 60.l/2

{*AI95-00195-01*} Clarified that the default implementation of S'Input does normal initialization on the object that it passes to S'Read. 60.m/2

{*AI95-00195-01*} Explicitly stated that what is read from a stream when a required check fails is unspecified. 60.n/2

{*AI95-00251-01*} Defined availability and default implementations for types with progenitors. 60.o/2

{*AI95-00279-01*} Specified that Constraint_Error is raised if the internal tag retrieved for S'Class'Input is for some type not covered by S'Class or is abstract. We also explicitly state that the program is erroneous if the tag has not been created or does not currently exist in the partition. (Ada 95 did not specify what happened in these cases; it's very unlikely to have provided some useful result, so this is not considered an inconsistency.) 60.p/2

{*AI95-00344-01*} Added wording to support nested type extensions. S'Input and S'Output always raise Tag_Error for such extensions, and such extensions were not permitted in Ada 95, so this is neither an extension nor an incompatibility. 60.q/2

{*AI95-00366-01*} Defined *supports external streaming* to put all of the rules about “good” stream attributes in one place. This is used for distribution and for defining pragma Pure. 60.r/2

{*AI95-00441-01*} Added the **not null** qualifier to the first parameter of all of the stream attributes, so that the semantics doesn't change between Ada 95 and Ada 2005. This change is compatible, because mode conformance is required for subprograms specified as stream attributes, and null_exclusions are not considered for mode conformance. 60.s/2

{*AI95-00444-01*} Improved the wording to make it clear that we don't define the default implementations of attributes that cannot be called (that is, aren't “available”). Also clarified when inheritance takes place. 60.t/2

Incompatibilities With Ada 2005

{*AI05-0039-1*} **Correction:** Added a requirement that stream attributes be specified by a static subprogram name rather than a dynamic expression. Expressions cannot provide any useful functionality because of the freezing rules, and the possibility of them complicates implementations. Only pathological programs should be affected. 60.u/3

Extensions to Ada 2005

60.v/3 {*AI05-0007-1*} **Correction:** Stream attributes for scalar types can be specified with subprograms that take the first subtype as well as the base type. This eliminates confusion about which subtype is appropriate for attributes specified for partial views whose full type is a scalar type. It also eliminates a common user error (forgetting 'Base).

Wording Changes from Ada 2005

60.w/3 {*AI05-0023-1*} **Correction:** Corrected the definition of the default version S'Read and S'Input to be well-defined if S is a discriminated type with defaulted discriminants and some components require initialization and/or finalizations.

60.x/3 {*AI05-0065-1*} **Correction:** Defined remote access types to support external streaming, since that is their purpose.

60.y/3 {*AI05-0109-1*} **Correction:** Removed a misleading phrase which implies that Constraint_Error is raised for internal tags of the wrong type, when Tag_Error should be raised for such tags.

60.z/3 {*AI05-0139-2*} Clarified that arrays with convention Fortran are written in column-major order, rather then row-major order. This is necessary in order that streaming of Fortran arrays is efficient.

60.aa/3 {*AI05-0192-1*} **Correction:** Clarified that the profile of an inherited stream attribute is as defined for an inherited primitive subprogram, while the default implementation of the same attribute might have a different profile.

60.bb/3 {*AI05-0194-1*} **Correction:** Clarified that Stream_Size has no effect on and is not effected by user-defined stream attributes.

Inconsistencies With Ada 2012

60.cc/5 {*AI12-0419-1*} The switch from inheritance to “implicitly composed” aspects means that there exists an exceedingly unlikely case where Ada 2022 code will raise Constraint_Error where Ada 2012 code would not. That case requires deriving a constrained untagged composite type *C* from an unconstrained parent type *P*, using a type conversion to *P* of the Input attribute of *C*, **and** streaming in a value that violates the constraints of *C*. Simply using the Input attribute of *P* instead of that of *C* will restore the Ada 2012 behavior.

Extensions to Ada 2012

60.dd/4 {*AI12-0106-1*} **Corrigendum:** Defined how to specify a class-wide stream-oriented attribute using an aspect_specification. It was always intended that this was possible, but the method was not clear, as a class-wide type never has an explicit declaration.

Wording Changes from Ada 2012

60.ee/4 {*AI12-0030-1*} **Corrigendum:** Defined the runtime effect of stream attributes for untagged limited types, as there is a weird corner case where they can be called. We don't specify this as an inconsistency, as it doesn't make semantic sense to stream a task, and nothing useful could have been done with that, so it should not exist in any programs.

60.ff/4 {*AI12-0106-1*} **Corrigendum:** Clarified that the same Legality Rules apply when a stream-oriented attribute is specified via an aspect_specification as applied when it is specified via an attribute_definition_clause.

60.gg/5 {*AI12-0064-2*} {*AI12-0396-1*} Added the definition of Nonblocking (see 9.5) and Global (see 6.1.2) for stream-oriented attributes.

60.hh/5 {*AI12-0419-1*} Revised to define composition of stream-oriented attributes for untagged derived types in order to avoid using inheritance for these attributes/aspects.

13.14 Freezing Rules

1/3 {*AI05-0299-1*} [This subclause defines a place in the program text where each declared entity becomes “frozen”. A use of an entity, such as a reference to it by name, or (for a type) an expression of the type, causes freezing of the entity in some contexts, as described below. The Legality Rules forbid certain kinds of uses of an entity in the region of text where it is frozen.]

1.a **Reason:** This concept has two purposes: a compile-time one and a run-time one.

1.b The compile-time purpose of the freezing rules comes from the fact that the evaluation of static expressions depends on overload resolution, and overload resolution sometimes depends on the value of a static expression. (The dependence of static evaluation upon overload resolution is obvious. The dependence in the other direction is more subtle. There are three rules that require static expressions in contexts that can appear in declarative places: The expression in an attribute_designator shall be static. In a record aggregate, variant-controlling discriminants shall be static. In an array aggregate with more than one named association, the choices shall be static. The compiler needs to know the value of

these expressions in order to perform overload resolution and legality checking.) We wish to allow a compiler to evaluate static expressions when it sees them in a single pass over the compilation_unit. The freezing rules ensure that.

The run-time purpose of the freezing rules is called the “linear elaboration model”. This means that declarations are elaborated in the order in which they appear in the program text, and later elaborations can depend on the results of earlier ones. The elaboration of the declarations of certain entities requires run-time information about the implementation details of other entities. The freezing rules ensure that this information has been calculated by the time it is used. For example, suppose the initial value of a constant is the result of a function call that takes a parameter of type *T*. In order to pass that parameter, the size of type *T* has to be known. If *T* is composite, that size might be known only at run time. 1.c

(Note that in these discussions, words like “before” and “after” generally refer to places in the program text, as opposed to times at run time.) 1.d

Discussion: The “implementation details” we're talking about above are: 1.e

- For a tagged type, the implementations of all the primitive subprograms of the type — that is (in the canonical implementation model), the contents of the type descriptor, which contains pointers to the code for each primitive subprogram. 1.f
- For a type, the full type declaration of any parts (including the type itself) that are private. 1.g
- For a deferred constant, the full constant declaration, which gives the constant's value. (Since this information necessarily comes after the constant's type and subtype are fully known, there's no need to worry about its type or subtype.) 1.h
- For any entity, representation information specified by the user via representation items. Most representation items are for types or subtypes; however, various other kinds of entities, such as objects and subprograms, are possible. 1.i

{*AI05-0005-1*} Similar issues arise for incomplete types. However, we do not use freezing to prevent premature access; incomplete types have different, more severe, restrictions. Similar issues also arise for subprograms, protected operations, tasks and generic units. However, we do not use freezing to prevent premature access for those, either; 3.11 prevents problems with run-time Elaboration_Checks. Even so, freezing is used for these entities to prevent giving representation items too late (that is, after uses that require representation information, such as calls). 1.j/3

Language Design Principles

An evaluable construct should freeze anything that's needed to evaluate it. 1.k

However, if the construct is not evaluated where it appears, let it cause freezing later, when it is evaluated. This is the case for default_expressions and default_names. (Formal parameters, generic formal parameters, and components can have default_expressions or default_names.) 1.l

The compiler should be allowed to evaluate static expressions without knowledge of their context. (I.e. there should not be any special rules for static expressions that happen to occur in a context that requires a static expression.) 1.m

Compilers should be allowed to evaluate static expressions (and record the results) using the run-time representation of the type. For example, suppose Color'Pos(Red) = 1, but the internal code for Red is 37. If the value of a static expression is Red, some compilers might store 1 in their symbol table, and other compilers might store 37. Either compiler design should be feasible. 1.n

Compilers should never be required to detect erroneousness or exceptions at compile time (although it's very nice if they do). This implies that we should not require code-generation for a nonstatic expression of type *T* too early, even if we can prove that that expression will be erroneous, or will raise an exception. 1.o

Here's an example (modified from AI83-00039, Example 3): 1.p

```
type T is                                                    1.q
    record
        ...
    end record;
function F return T;
function G(X : T) return Boolean;
Y : Boolean := G(F); -- doesn't force T in Ada 83
for T use
    record
        ...
    end record;
```

AI83-00039 says this is legal. Of course, it raises Program_Error because the function bodies aren't elaborated yet. A one-pass compiler has to generate code for an expression of type T before it knows the representation of T. Here's a similar example, which AI83-00039 also says is legal: 1.r

1.s
```
package P is
    type T is private;
    function F return T;
    function G(X : T) return Boolean;
    Y : Boolean := G(F); -- doesn't force T in Ada 83
private
    type T is
        record
            ...
        end record;
end P;
```

1.t If T's size were dynamic, that size would be stored in some compiler-generated dope; this dope would be initialized at the place of the full type declaration. However, the generated code for the function calls would most likely allocate a temp of the size specified by the dope *before* checking for Program_Error. That dope would contain uninitialized junk, resulting in disaster. To avoid doing that, the compiler would have to determine, at compile time, that the expression will raise Program_Error.

1.u This is silly. If we're going to require compilers to detect the exception at compile time, we might as well formulate the rule as a legality rule.

1.v Compilers should not be required to generate code to load the value of a variable before the address of the variable has been determined.

1.w After an entity has been frozen, no further requirements may be placed on its representation (such as by a representation item or a full_type_declaration).

2 The *freezing* of an entity occurs at one or more places (*freezing points*) in the program text where the representation for the entity has to be fully determined. Each entity is frozen from its first freezing point to the end of the program text (given the ordering of compilation units defined in 10.1.4).

2.a **Ramification:** The “representation” for a subprogram includes its calling convention and means for referencing the subprogram body, either a “link-name” or specified address. It does not include the code for the subprogram body itself, nor its address if a link-name is used to reference the body.

2.1/3 {*AI05-0019-1*} {*AI05-0299-1*} This subclause also defines a place in the program text where the profile of each declared callable entity becomes *frozen*. A use of a callable entity causes freezing of its profile in some contexts, as described below. At the place where the profile of a callable entity becomes frozen, the entity itself becomes frozen.

3/5 {*8652/0014*} {*AI05-0017-1*} {*AI05-0019-1*} {*AI12-0103-1*} {*AI12-0155-1*} {*AI12-0168-1*} {*AI12-0373-1*} The end of a declarative_part, protected_body, or a declaration of a library package or generic library package, causes *freezing* of each entity and profile declared within it, as well as the entity itself in the case of the declaration of a library unit. A noninstance proper_body, body_stub, or entry_body causes freezing of each entity and profile declared before it within the same declarative_part.

3.a **Discussion:** This is worded carefully to handle nested packages and private types. Entities declared in a nested package_specification will be frozen by some containing construct.

3.b/5 {*AI05-0017-1*} {*AI12-0155-1*} An incomplete type declared in the private part of a library package_specification can be completed in the body. For other incomplete types (and in the bodies of library packages), the completion of the type will be frozen at the end of the package or declarative_part.

3.c/3 **Ramification:** {*AI05-0229-1*} The part about bodies does not say *immediately* within. A renaming-as-body does not have this property. Nor does an imported body

3.d/5 **Reason:** {*AI12-0300-1*} The reason bodies cause freezing is because we want proper_bodies and body_stubs to be interchangeable — one should be able to move a proper_body to a subunit, and vice versa, without changing the semantics. Clearly, anything that should cause freezing should do so even if it's inside a proper_body. However, if we make it a body_stub, then the compiler can't see that thing that should cause freezing. So we make body_stubs cause freezing, just in case they contain something that should cause freezing. But that means we need to do the same for proper_bodies.

3.e Another reason for bodies to cause freezing, there could be an added implementation burden if an entity declared in an enclosing declarative_part is frozen within a nested body, since some compilers look at bodies after looking at the containing declarative_part.

3.f/4 {*AI05-0177-1*} {*AI12-0103-1*} Note that null_procedure_declarations and expression_function_declarations (even when those are used as completions), as well as generic_instantiations and renames-as-bodies do not necessarily cause freezing; each have their own specific rules.

Ramification: {*AI12-0103-1*} {*AI12-0005-1*} Note that the rule about proper bodies causing freezing only applies in declarative_parts. All of the kinds of bodies (see 3.11.1 – keep in mind the difference from bodys) that are allowed in a package specification have their own freezing rules, so they don't need to be covered by the above rule. 3.g/5

{*8652/0046*} {*AI95-00106-01*} A construct that (explicitly or implicitly) references an entity can cause the *freezing* of the entity, as defined by subsequent paragraphs. At the place where a construct causes freezing, each name, expression, implicit_dereference[, or range] within the construct causes freezing: 4/1

Ramification: Note that in the sense of this paragraph, a subtype_mark "references" the denoted subtype, but not the type. 4.a

- {*AI05-0213-1*} The occurrence of a generic_instantiation causes freezing, except that a name which is a generic actual parameter whose corresponding generic formal parameter is a formal incomplete type (see 12.5.1) does not cause freezing. In addition, if a parameter of the instantiation is defaulted, the default_expression or default_name for that parameter causes freezing. 5/3

  Ramification: {*AI05-0213-1*} Thus, an actual parameter corresponding to a formal incomplete type parameter may denote an incomplete or private type which is not completely defined at the point of the generic_instantiation. 5.a/3

- {*AI12-0103-1*} {*AI12-0157-1*} At the occurrence of an expression_function_declaration that is a completion, the return expression of the expression function causes freezing. 5.1/4

  Reason: {*AI12-0103-1*} This rule prevents calls through access values to an expression that might have unfrozen parts. Typically, elaboration checks and other freezing rules prevent this, but in this case the completion is elaborated and since this is not a body it does not by itself freeze anything that precedes it. 5.b/4

- {*AI12-0132-1*} {*AI12-0157-1*} At the occurrence of a renames-as-body whose *callable_entity_*name denotes an expression function, the return expression of the expression function causes freezing. 5.2/4

- The occurrence of an object_declaration that has no corresponding completion causes freezing. 6

  Ramification: Note that this does not include a formal_object_declaration. 6.a

- The declaration of a record extension causes freezing of the parent subtype. 7

  Ramification: This combined with another rule specifying that primitive subprogram declarations shall precede freezing ensures that all descendants of a tagged type implement all of its dispatching operations. 7.a

  {*AI95-00251-01*} The declaration of a private extension does not cause freezing. The freezing is deferred until the full type declaration, which will necessarily be for a record extension, task, or protected type (the latter only for a limited private extension derived from an interface). 7.b/2

- {*AI95-00251-01*} The declaration of a record extension, interface type, task unit, or protected unit causes freezing of any progenitor types specified in the declaration. 7.1/2

  Reason: This rule has the same purpose as the one above: ensuring that all descendants of an interface tagged type implement all of its dispatching operations. As with the previous rule, a private extension does not freeze its progenitors; the full type declaration (which must have the same progenitors) will do that. 7.b.1/2

  Ramification: An interface type can be a parent as well as a progenitor; these rules are similar so that the location of an interface in a record extension does not have an effect on the freezing of the interface type. 7.b.2/2

- {*AI05-0183-1*} {*AI12-0396-1*} {*AI12-0407-1*} {*AI12-0439-1*} At the freezing point of the entity associated with an aspect_specification, any static expressions within the aspect_specification cause freezing, as do expressions or names in aspect_definitions for representation aspects, or operational aspects that have a corresponding operational attribute. Similarly, if an aspect_definition for an operational aspect, other than an assertion aspect, can affect the Name Resolution, Static Semantics, or Legality Rules of a subsequent construct, then any expressions or names within the aspect_definition cause freezing at the freezing point of the associated entity. Any static expressions within an aspect_specification also cause freezing at the end of the immediately enclosing declaration list. For the purposes of this rule, if there is no declared entity associated with an aspect_specification, the freezing point is considered to occur immediately following the aspect_specification. 7.2/5

8/4 {*8652/0046*} {*AI95-00106-01*} {*AI05-0177-1*} {*AI05-0183-1*} {*AI05-0157-1*} A static expression (other than within an aspect_specification) causes freezing where it occurs. An object name or nonstatic expression causes freezing where it occurs, unless the name or expression is part of a default_expression, a default_name, the return expression of an expression function, an aspect_specification, or a per-object expression of a component's constraint, in which case, the freezing occurs later as part of another construct or at the freezing point of an associated entity.

8.1/3 {*8652/0046*} {*AI95-00106-01*} {*AI05-0019-1*} An implicit call freezes the same entities and profiles that would be frozen by an explicit call. This is true even if the implicit call is removed via implementation permissions.

8.2/1 {*8652/0046*} {*AI95-00106-01*} If an expression is implicitly converted to a type or subtype *T*, then at the place where the expression causes freezing, *T* is frozen.

9 The following rules define which entities are frozen at the place where a construct causes freezing:

10/5 - {*AI12-0367-1*} At the place where an expression causes freezing, the type of the expression is frozen, unless the expression is an enumeration literal used as a discrete_choice of the array_aggregate of an enumeration_representation_clause or as the aspect_definition of a specification for aspect Default_Value.

10.a **Reason:** We considered making enumeration literals never cause freezing, which would be more upward compatible, but examples like the variant record aggregate (Discrim => Red, ...) caused us to change our mind. Furthermore, an enumeration literal is a static expression, so the implementation should be allowed to represent it using its representation.

10.a.1/5 {*AI12-0367-1*} We exclude uses of enumeration literals that are used in part for setting the representation of the type (aspect Default_Value is defined to be a representation aspect) so we don't freeze the type while determining its representation.

10.b **Ramification:** The following pathological example was legal in Ada 83, but is illegal in Ada 95:

10.c
```
package P1 is
    type T is private;
    package P2 is
        type Composite(D : Boolean) is
            record
                case D is
                    when False => Cf : Integer;
                    when True  => Ct : T;
                end case;
            end record;
    end P2;
    X : Boolean := P2."="( (False,1), (False,1) );
private
    type T is array(1..Func_Call) of Integer;
end;
```

10.d In Ada 95, the declaration of X freezes Composite (because it contains an expression of that type), which in turn freezes T (even though Ct does not exist in this particular case). But type T is not completely defined at that point, violating the rule that a type shall be completely defined before it is frozen. In Ada 83, on the other hand, there is no occurrence of the name T, hence no forcing occurrence of T.

10.1/4 - {*AI05-0019-1*} {*AI05-0177-1*} {*AI12-0157-1*} At the place where a function call causes freezing, the profile of the function is frozen. Furthermore, if a parameter of the call is defaulted, the default_expression for that parameter causes freezing. If the function call is to an expression function, the return expression of the expression function causes freezing.

10.e/3 **Reason:** {*AI05-0019-1*} This is the important rule for profile freezing: a call freezes the profile. That's because generating the call will need to know how the parameters are passed, and that will require knowing details of the types. Other uses of subprograms do not need to know about the parameters, and thus only freeze the subprogram, and not the profile.

10.f/3 Note that we don't need to consider procedure or entry calls, since a body freezes everything that precedes it, and the end of a declarative part freezes everything in the declarative part.

10.g/4 **Ramification:** {*AI05-0177-1*} {*AI12-0103-1*} {*AI12-0157-1*} Freezing of the return expression of an expression function only needs to be considered when the expression function is in the same compilation unit and there are no

intervening bodys; the end of a declarative_part or library package freezes everything in it, and a body freezes everything declared before it.

- {*AI05-0019-1*} {*AI05-0177-1*} {*AI05-0296-1*} {*AI12-0157-1*} At the place where a generic_instantiation causes freezing of a callable entity, the profile of that entity is frozen unless the formal subprogram corresponding to the callable entity has a parameter or result of a formal untagged incomplete type; if the callable entity is an expression function, the return expression of the expression function causes freezing. 10.2/4

  Reason: Elaboration of the generic might call the actual for one of its formal subprograms, so we need to know the profile and (for an expression function) expression. 10.h/3

- {*AI05-0177-1*} {*AI12-0157-1*} At the place where a use of the Access or Unchecked_Access attribute whose prefix denotes an expression function causes freezing, the return expression of the expression function causes freezing. 10.3/4

  Reason: This is needed to avoid calls to unfrozen expressions. Consider: 10.i/3

  ```
  package Pack is                                                                          10.j/3
     type Flub is range 0 .. 100;                                                          10.k/3
     function Foo (A : in Natural) return Natural is                                       10.l/3
        (A + Flub'Size); -- The expression is not frozen here.
  {AI12-0005-1}    type Bar is access function (A : in Natural) return Natural;          10.m/4
     P : Bar := Foo'Access; -- (A)                                                         10.n/3
     Val : Natural := P.all(5); -- (B)                                                     10.o/3
  end Pack;                                                                              10.p/3
  ```

  If point (A) did not freeze the expression of Foo (which freezes Flub), then the call at point (B) would be depending on the aspects of the unfrozen type Flub. That would be bad. 10.q/3

- At the place where a name causes freezing, the entity denoted by the name is frozen, unless the name is a prefix of an expanded name; at the place where an object name causes freezing, the nominal subtype associated with the name is frozen. 11

  Ramification: {*AI95-00114-01*} This only matters in the presence of deferred constants or access types; an object_declaration other than a deferred constant declaration causes freezing of the nominal subtype, plus all component junk. 11.a/2

  This paragraph was deleted.{*8652/0046*} {*AI95-00106-01*} 11.b/1

- {*8652/0046*} {*AI95-00106-01*} At the place where an implicit_dereference causes freezing, the nominal subtype associated with the implicit_dereference is frozen. 11.1/1

  Discussion: This rule ensures that X.D freezes the same entities that X.**all**.D does. Note that an implicit_dereference is neither a name nor expression by itself, so it isn't covered by other rules. 11.c/2

- [At the place where a range causes freezing, the type of the range is frozen.] 12

  Proof: This is consequence of the facts that expressions freeze their type, and the Range attribute is defined to be equivalent to a pair of expressions separated by "..".} 12.a

- At the place where an allocator causes freezing, the designated subtype of its type is frozen. If the type of the allocator is a derived type, then all ancestor types are also frozen. 13

  Ramification: Allocators also freeze the named subtype, as a consequence of other rules. 13.a

  The ancestor types are frozen to prevent things like this: 13.b

  ```
  type Pool_Ptr is access System.Storage_Pools.Root_Storage_Pool'Class;                  13.c
  function F return Pool_Ptr;

  package P is                                                                           13.d
      type A1 is access Boolean;
      type A2 is new A1;
      type A3 is new A2;
      X : A3 := new Boolean; -- Don't know what pool yet!
      for A1'Storage_Pool use F.all;
  end P;
  ```

  This is necessary because derived access types share their parent's pool. 13.e

14/3 • {*AI05-0019-1*} At the place where a profile is frozen, each subtype of the profile is frozen. If the corresponding callable entity is a member of an entry family, the index subtype of the family is frozen.

14.a/3 *This paragraph was deleted.*

15/5 • {*AI12-0186-1*} At the place where a subtype is frozen, its type is frozen. At the place where a type is frozen, any expressions or names within the full type definition cause freezing, other than those that occur within an access_type_definition or an access_definition; the first subtype, and any component subtypes, index subtypes, and parent subtype of the type are frozen as well. For a specific tagged type, the corresponding class-wide type is frozen as well. For a class-wide type, the corresponding specific type is frozen as well.

15.a **Ramification:** Freezing a type needs to freeze its first subtype in order to preserve the property that the subtype-specific aspects of statically matching subtypes are the same.

15.b Freezing an access type does not freeze its designated subtype.

15.1/3 • {*AI95-00341-01*} {*AI05-0019-1*} At the place where a specific tagged type is frozen, the primitive subprograms of the type are frozen. At the place where a type is frozen, any subprogram named in an attribute_definition_clause for the type is frozen.

15.c/2 **Reason:** We have a language design principle that all of the details of a specific tagged type are known at its freezing point. But that is only true if the primitive subprograms are frozen at this point as well. Late changes of Import and address clauses violate the principle.

15.d/2 **Implementation Note:** This rule means that no implicit call to Initialize or Adjust can freeze a subprogram (the type and thus subprograms would have been frozen at worst at the same point).

15.e/3 **Discussion:** {*AI05-0019-1*} The second sentence is the rule that makes it possible to check that only subprograms with convention Ada are specified in attribute_definition_clauses without jumping through hoops.

15.2/5 • {*AI12-0396-1*} At the place where a construct causes freezing, if the construct includes a check associated with some assertion aspect [(independent of whether the check is enabled)], or depends on the definition of some operational aspect as part of its Dynamic Semantics, any names or expressions in the aspect_definition for the aspect cause freezing.

15.f/5 **Ramification:** Aspects such as Constant_Indexing cause freezing when they are used — the name in a Constant_Indexing would cause freezing when a generalized_indexing that makes use of it causes freezing.

15.3/5 {*AI12-0155-1*} Notwithstanding the rest of this subclause, freezing an incomplete view has no effect.

15.g/5 **Ramification:** In particular, freezing an incomplete view does not freeze the completion of the view. For an incomplete type, once the completion of the incomplete type is seen, any further references to the type will be of the complete type, which will be frozen in the normal way. Legality Rules prevent all problems for incomplete types themselves.

15.h/5 In contrast, freezing a partial view does have an effect (the completion of the view is frozen and the program is illegal if it hasn't occurred yet).

Legality Rules

16 [The explicit declaration of a primitive subprogram of a tagged type shall occur before the type is frozen (see 3.9.2).]

16.a **Reason:** This rule is needed because (1) we don't want people dispatching to things that haven't been declared yet, and (2) we want to allow tagged type descriptors to be static (allocated statically, and initialized to link-time-known symbols). Suppose T2 inherits primitive P from T1, and then overrides P. Suppose P is called *before* the declaration of the overriding P. What should it dispatch to? If the answer is the new P, we've violated the first principle above. If the answer is the old P, we've violated the second principle. (A call to the new one necessarily raises Program_Error, but that's beside the point.)

16.b Note that a call upon a dispatching operation of type *T* will freeze *T*.

16.c We considered applying this rule to all derived types, for uniformity. However, that would be upward incompatible, so we rejected the idea. As in Ada 83, for an untagged type, the above call upon P will call the old P (which is arguably confusing).

To be honest: {*AI05-0222-1*} This rule only applies to "original" declarations and not to the completion of a primitive subprogram, even though a completion is technically an explicit declaration, and it may declare a primitive subprogram. 16.d/3

[A type shall be completely defined before it is frozen (see 3.11.1 and 7.3).] 17

[The completion of a deferred constant declaration shall occur before the constant is frozen (see 7.4).] 18

This paragraph was deleted.{*AI95-00114-01*} {*AI05-0299-1*} {*AI12-0181-1*} 18.a/5

{*8652/0009*} {*AI95-00137-01*} [An operational or representation item that directly specifies an aspect of an entity shall appear before the entity is frozen (see 13.1).] 19/1

Discussion: {*8652/0009*} {*AI95-00137-01*} From RM83-13.1(7). The wording here forbids freezing within the aspect_clause itself, which was not true of the Ada 83 wording. The wording of this rule is carefully written to work properly for type-related representation items. For example, an enumeration_representation_clause is illegal after the type is frozen, even though the _clause refers to the first subtype. 19.a/1

This paragraph was deleted.{*AI95-00114-01*} {*AI12-0181-1*} 19.a.1/5

Proof: {*AI95-00114-01*} {*AI12-0181-1*} The above Legality Rules are stated “officially” in the referenced subclauses. 19.b/5

Discussion: Here's an example that illustrates when freezing occurs in the presence of defaults: 19.c

```
type T is ...;                                                    19.d
function F return T;
type R is
    record
        C : T := F;
        D : Boolean := F = F;
    end record;
X : R;
```

Since the elaboration of R's declaration does not allocate component C, there is no need to freeze C's subtype at that place. Similarly, since the elaboration of R does not evaluate the default_expression “F = F”, there is no need to freeze the types involved at that point. However, the declaration of X *does* need to freeze these things. Note that even if component C did not exist, the elaboration of the declaration of X would still need information about T — even though D is not of type T, its default_expression requires that information. 19.e

Ramification: Although we define freezing in terms of the program text as a whole (i.e. after applying the rules of Clause 10), the freezing rules actually have no effect beyond compilation unit boundaries. 19.f

Reason: That is important, because Clause 10 allows some implementation definedness in the order of things, and we don't want the freezing rules to be implementation defined. 19.g

Ramification: These rules also have no effect in statements — they only apply within a single declarative_part, package_specification, task_definition, protected_definition, or protected_body. 19.h

Implementation Note: An implementation may choose to generate code for default_expressions and default_names in line at the place of use. Alternatively, an implementation may choose to generate thunks (subprograms implicitly generated by the compiler) for evaluation of defaults. Thunk generation cannot, in general, be done at the place of the declaration that includes the default. Instead, they can be generated at the first freezing point of the type(s) involved. (It is impossible to write a purely one-pass Ada compiler, for various reasons. This is one of them — the compiler needs to store a representation of defaults in its symbol table, and then walk that representation later, no earlier than the first freezing point.) 19.i

In implementation terms, the linear elaboration model can be thought of as preventing uninitialized dope. For example, the implementation might generate dope to contain the size of a private type. This dope is initialized at the place where the type becomes completely defined. It cannot be initialized earlier, because of the order-of-elaboration rules. The freezing rules prevent elaboration of earlier declarations from accessing the size dope for a private type before it is initialized. 19.j

{*AI05-0299-1*} 2.8 overrides the freezing rules in the case of unrecognized pragmas; similarly 13.1.1 overrides the freezing rules in the case of a specification of an unrecognized aspect. 19.k/3

{*8652/0009*} {*AI95-00137-01*} An aspect_clause for an entity should most certainly *not* be a freezing point for the entity. 19.l/1

Dynamic Semantics

{*AI95-00279-01*} The tag (see 3.9) of a tagged type T is created at the point where T is frozen. 20/2

Incompatibilities With Ada 83

20.a RM83 defines a forcing occurrence of a type as follows: “A forcing occurrence is any occurrence [of the name of the type, subtypes of the type, or types or subtypes with subcomponents of the type] other than in a type or subtype declaration, a subprogram specification, an entry declaration, a deferred constant declaration, a pragma, or a representation_clause for the type itself. In any case, an occurrence within an expression is always forcing.”

20.b It seems like the wording allows things like this:

20.c
```
type A is array(Integer range 1..10) of Boolean;
subtype S is Integer range A'Range;
    -- not forcing for A
```

20.d Occurrences within pragmas can cause freezing in Ada 95. (Since such pragmas are ignored in Ada 83, this will probably fix more bugs than it causes.)

Extensions to Ada 83

20.e In Ada 95, generic_formal_parameter_declarations do not normally freeze the entities from which they are defined. For example:

20.f
```
package Outer is
    type T is tagged limited private;
    generic
        type T2 is
            new T with private;  -- Does not freeze T
                                 -- in Ada 95.
    package Inner is
        ...
    end Inner;
private
    type T is ...;
end Outer;
```

20.g This is important for the usability of generics. The above example uses the Ada 95 feature of formal derived types. Examples using the kinds of formal parameters already allowed in Ada 83 are well known. See, for example, comments 83-00627 and 83-00688. The extensive use expected for formal derived types makes this issue even more compelling than described by those comments. Unfortunately, we are unable to solve the problem that explicit_generic_actual_parameters cause freezing, even though a package equivalent to the instance would not cause freezing. This is primarily because such an equivalent package would have its body in the body of the containing program unit, whereas an instance has its body right there.

Wording Changes from Ada 83

20.h The concept of freezing is based on Ada 83's concept of “forcing occurrences”. The first freezing point of an entity corresponds roughly to the place of the first forcing occurrence, in Ada 83 terms. The reason for changing the terminology is that the new rules do not refer to any particular “occurrence” of a name of an entity. Instead, we refer to “uses” of an entity, which are sometimes implicit.

20.i In Ada 83, forcing occurrences were used only in rules about representation_clauses. We have expanded the concept to cover private types, because the rules stated in RM83-7.4.1(4) are almost identical to the forcing occurrence rules.

20.j The Ada 83 rules are changed in Ada 95 for the following reasons:

20.k
- The Ada 83 rules do not work right for subtype-specific aspects. In an earlier version of Ada 9X, we considered allowing representation items to apply to subtypes other than the first subtype. This was part of the reason for changing the Ada 83 rules. However, now that we have dropped that functionality, we still need the rules to be different from the Ada 83 rules.

20.l
- The Ada 83 rules do not achieve the intended effect. In Ada 83, either with or without the AIs, it is possible to force the compiler to generate code that references uninitialized dope, or force it to detect erroneousness and exception raising at compile time.

20.m
- It was a goal of Ada 83 to avoid uninitialized access values. However, in the case of deferred constants, this goal was not achieved.

20.n
- The Ada 83 rules are not only too weak — they are also too strong. They allow loopholes (as described above), but they also prevent certain kinds of default_expressions that are harmless, and certain kinds of generic_declarations that are both harmless and very useful.

20.o/2
- {*AI95-00114-01*} Ada 83 had a case where an aspect_clause had a strong effect on the semantics of the program — 'Small. This caused certain semantic anomalies. There are more cases in Ada 95, because the attribute_definition_clause has been generalized.

Incompatibilities With Ada 95

{*8652/0046*} {*AI95-00106-01*} {*AI95-00341-01*} **Corrigendum:** Various freezing rules were added to fix holes in the rules. Most importantly, implicit calls are now freezing, which make some representation clauses illegal in Ada 2005 that were legal (but dubious) in Ada 95. **Amendment Correction:** Similarly, the primitive subprograms of a specific tagged type are frozen when the type is frozen, preventing dubious convention changes (and address clauses) after the freezing point. In both cases, the code is dubious and the workaround is easy. 20.p/2

Wording Changes from Ada 95

{*8652/0009*} {*AI95-00137-01*} **Corrigendum:** Added wording to specify that both operational and representation attributes must be specified before the type is frozen. 20.q/2

{*AI95-00251-01*} Added wording that declaring a specific descendant of an interface type freezes the interface type. 20.r/2

{*AI95-00279-01*} Added wording that defines when a tag is created for a type (at the freezing point of the type). This is used to specify checking for uncreated tags (see 3.9). 20.s/2

Incompatibilities With Ada 2005

{*AI05-0019-1*} **Correction:** Separated the freezing of the profile from the rest of a subprogram, in order to reduce the impact of the Ada 95 incompatibility noted above. (The effects were much more limiting than expected.) 20.t/3

Wording Changes from Ada 2005

{*AI05-0017-1*} **Correction:** Reworded so that incomplete types with a deferred completion aren't prematurely frozen. 20.u/3

{*AI05-0177-1*} Added freezing rules for expression functions; these are frozen at the point of call, not the point of declaration, like default expressions. 20.v/3

{*AI05-0183-1*} Added freezing rules for aspect_specifications; these are frozen at the freezing point of the associated entity, not the point of declaration. 20.w/3

{*AI05-0213-1*} Added freezing rules for formal incomplete types; the corresponding actual is not frozen. 20.x/3

Wording Changes from Ada 2012

{*AI12-0103-1*} {*AI12-0132-1*} **Corrigendum:** Clarified when and what an expression_function_declaration that is a completion or that is the target of a renames-as-body freezes. This is formally an incompatibility, but as all known implementations freeze expression functions more aggressively than allowed by either the old or new wording, practically this will be an extension. 20.y/4

{*AI12-0168-1*} **Correction:** Clarified that the implicit body created by an instance doesn't have any freezing effect. This was an unintentional change caused by the Technical Corrigendum 1 oversimplifying wording. 20.z/5

{*AI12-0155-1*} **Correction:** Clarified that incomplete types do not freeze in any case. The one known case where incomplete freezing was necessary has been plugged with a new Legality Rules (see 3.10.1). This matches the original intent (see Language Design Principles above) and eliminates whack-a-mole trying to allow such freezing without it having (almost) any effects. 20.aa/5

{*AI12-0186-1*} **Correction:** Clarified that the subtype names in an access type are not frozen at the point of the type declaration. This has always been true (there is an Ada 95-era AARM note that says so) but it didn't follow from the actual wording. Since the ACATS requires the AARM note to be true, no compiler could actually get this wrong, so no incompatibility is possible. 20.bb/5

{*AI12-0367-1*} **Correction:** Added a freezing exclusion for enumeration literals in the expression of a Default_Value aspect; otherwise, one could not set a default value for an enumeration type. 20.cc/5

{*AI12-0373-1*} Added missing definition of freezing of library units. 20.dd/5

{*AI12-0396-1*} **Correction:** Improved and clarified the freezing rules for aspect_specifications; these now depend on the kind of aspect. 20.ee/5

The Standard Libraries

Annex A
(normative)
Predefined Language Environment

[This Annex contains the specifications of library units that shall be provided by every implementation. There are three root library units: Ada, Interfaces, and System; other library units are children of these:] 1

{*8652/0047*} {*AI95-00081-01*} {*AI95-00424-01*} {*AI05-0001-1*} {*AI05-0049-1*} {*AI05-0069-1*} {*AI05-0111-3*} {*AI05-0136-1*} {*AI05-0137-1*} {*AI05-0166-1*} {*AI05-0168-1*} {*AI12-0021-1*} {*AI12-0208-1*} {*AI12-0234-1*} {*AI12-0254-1*} {*AI12-0293-1*} {*AI12-0321-1*} 2/5

[Standard — A.1
Ada — A.2
Assertions — 11.4.2
Asynchronous_Task_Control — D.11
Calendar — 9.6
Arithmetic — 9.6.1
Formatting — 9.6.1
Time_Zones — 9.6.1
Characters — A.3.1
Conversions — A.3.4
Handling — A.3.2
Latin_1 — A.3.3
Command_Line — A.15
Complex_Text_IO — G.1.3
Containers — A.18.1
Bounded_Doubly_Linked_Lists — A.18.20
Bounded_Hashed_Maps — A.18.21
Bounded_Hashed_Sets — A.18.23
Bounded_Indefinite_Holders — A.18.32
Bounded_Multiway_Trees — A.18.25
Bounded_Ordered_Maps — A.18.22
Bounded_Ordered_Sets — A.18.24
Bounded_Priority_Queues — A.18.31
Bounded_Synchronized_Queues — A.18.29
Bounded_Vectors — A.18.19
Doubly_Linked_Lists — A.18.3
Generic_Array_Sort — A.18.26
Generic_Constrained_Array_Sort — A.18.26
Generic_Sort — A.18.26
Hashed_Maps — A.18.5
Hashed_Sets — A.18.8
Indefinite_Doubly_Linked_Lists — A.18.12
Indefinite_Hashed_Maps — A.18.13
Indefinite_Hashed_Sets — A.18.15

- Standard (*...continued*)
  - Ada (*...continued*)
    - Containers (*...continued*)
      - Indefinite_Holders — A.18.18
      - Indefinite_Multiway_Trees — A.18.17
      - Indefinite_Ordered_Maps — A.18.14
      - Indefinite_Ordered_Sets — A.18.16
      - Indefinite_Vectors — A.18.11
      - Multiway_Trees — A.18.10
      - Ordered_Maps — A.18.6
      - Ordered_Sets — A.18.9
      - Synchronized_Queue_Interfaces — A.18.27
      - Unbounded_Priority_Queues — A.18.30
      - Unbounded_Synchronized_Queues — A.18.28
      - Vectors — A.18.2
    - Decimal — F.2
    - Direct_IO — A.8.4
    - Directories — A.16
      - Hierarchical_File_Names — A.16.1
      - Information — A.16
    - Dispatching — D.2.1
      - EDF — D.2.6
      - Non_Preemptive — D.2.4
      - Round_Robin — D.2.5
    - Dynamic_Priorities — D.5.1
    - Environment_Variables — A.17
    - Exceptions — 11.4.1
    - Execution_Time — D.14
      - Group_Budgets — D.14.2
      - Interrupts — D.14.3
      - Timers — D.14.1
    - Finalization — 7.6
    - Float_Text_IO — A.10.9
    - Float_Wide_Text_IO — A.11
    - Float_Wide_Wide_Text_IO — A.11

- Standard (*...continued*)
  - Ada (*...continued*)
    - Integer_Text_IO — A.10.8
    - Integer_Wide_Text_IO — A.11
    - Integer_Wide_Wide_Text_IO — A.11
    - Interrupts — C.3.2
      - Names — C.3.2
    - IO_Exceptions — A.13
    - Iterator_Interfaces — 5.5.1
    - Locales — A.19 Numerics — A.5
      - Big_Numbers — A.5.5
        - Big_Integers — A.5.6
        - Big_Reals — A.5.7
      - Complex_Arrays — G.3.2
      - Complex_Elementary_Functions — G.1.2
      - Complex_Types — G.1.1
      - Discrete_Random — A.5.2
      - Elementary_Functions — A.5.1
      - Float_Random — A.5.2
      - Generic_Complex_Arrays — G.3.2
      - Generic_Complex_Elementary_Functions — G.1.2
      - Generic_Complex_Types — G.1.1
      - Generic_Elementary_Functions — A.5.1
      - Generic_Real_Arrays — G.3.1
      - Real_Arrays — G.3.1
    - Real_Time — D.8
      - Timing_Events — D.15
    - Sequential_IO — A.8.1
    - Storage_IO — A.9
    - Streams — 13.13.1
      - Storage_Streams — 13.13.1
        - Bounded_FIFO_Streams — 13.13.1
        - FIFO_Streams — 13.13.1
      - Stream_IO — A.12.1
    - Strings — A.4.1
      - Bounded — A.4.4
        - Equal_Case_Insensitive — A.4.10
        - Hash — A.4.9
        - Hash_Case_Insensitive — A.4.9
        - Less_Case_Insensitive — A.4.10
      - Equal_Case_Insensitive — A.4.10
      - Fixed — A.4.3
        - Equal_Case_Insensitive — A.4.10
        - Hash — A.4.9
        - Hash_Case_Insensitive — A.4.9
        - Less_Case_Insensitive — A.4.10

Standard (*...continued*)
Ada (*...continued*)
Strings (*...continued*)
Hash — A.4.9
Hash_Case_Insensitive — A.4.9
Less_Case_Insensitive — A.4.10
Maps — A.4.2
Constants — A.4.6
Text_Buffers — A.4.12
Bounded — A.4.12
Unbounded — A.4.12
Unbounded — A.4.5
Equal_Case_Insensitive — A.4.10
Hash — A.4.9
Hash_Case_Insensitive — A.4.9
Less_Case_Insensitive — A.4.10
UTF_Encoding — A.4.11
Conversions — A.4.11
Strings — A.4.11
Wide_Strings — A.4.11
Wide_Wide_Strings — A.4.11
Wide_Bounded — A.4.7
Wide_Equal_Case_Insensitive — A.4.7
Wide_Hash — A.4.7
Wide_Hash_Case_Insensitive — A.4.7
Wide_Equal_Case_Insensitive — A.4.7
Wide_Fixed — A.4.7
Wide_Equal_Case_Insensitive — A.4.7
Wide_Hash — A.4.7
Wide_Hash_Case_Insensitive — A.4.7
Wide_Hash — A.4.7
Wide_Hash_Case_Insensitive — A.4.7
Wide_Maps — A.4.7
Wide_Constants — A.4.7
Wide_Unbounded — A.4.7
Wide_Equal_Case_Insensitive — A.4.7
Wide_Hash — A.4.7
Wide_Hash_Case_Insensitive — A.4.7
Wide_Wide_Bounded — A.4.8
Wide_Wide_Equal_Case_Insensitive — A.4.8
Wide_Wide_Hash — A.4.8
Wide_Wide_Hash_Case_Insensitive — A.4.8

Standard (*...continued*)
Ada (*...continued*)
Strings (*...continued*)
Wide_Wide_Equal_Case_Insensitive — A.4.8
Wide_Wide_Fixed — A.4.8
Wide_Wide_Equal_Case_Insensitive — A.4.8
Wide_Wide_Hash — A.4.8
Wide_Wide_Hash_Case_Insensitive — A.4.8
Wide_Wide_Hash — A.4.8
Wide_Wide_Hash_Case_Insensitive — A.4.8
Wide_Wide_Maps — A.4.8
Wide_Wide_Constants — A.4.8
Wide_Wide_Unbounded — A.4.8
Wide_Wide_Equal_Case_Insensitive — A.4.8
Wide_Wide_Hash — A.4.8
Wide_Wide_Hash_Case_Insensitive — A.4.8
Synchronous_Barriers — D.10.1
Synchronous_Task_Control — D.10
EDF — D.10
Tags — 3.9
Generic_Dispatching_Constructor — 3.9
Task_Attributes — C.7.2
Task_Identification — C.7.1
Task_Termination — C.7.3
Text_IO — A.10.1
Bounded_IO — A.10.11
Complex_IO — G.1.3
Editing — F.3.3
Text_Streams — A.12.2
Unbounded_IO — A.10.12
Unchecked_Conversion — 13.9
Unchecked_Deallocate_Subpool — 13.11.5
Unchecked_Deallocation — 13.11.2
Wide_Characters — A.3.1
Handling — A.3.5
Wide_Command_Line — A.15.1
Wide_Directories — A.16.2
Wide_Environment_Variables — A.17.1

Standard (*...continued*)
- Ada (*...continued*)
  - Wide_Text_IO — A.11
    - Complex_IO — G.1.4
    - Editing — F.3.4
    - Text_Streams — A.12.3
    - Wide_Bounded_IO — A.11
    - Wide_Unbounded_IO — A.11
  - Wide_Wide_Characters — A.3.1
    - Handling — A.3.6
  - Wide_Wide_Command_Line — A.15.1
  - Wide_Wide_Directories — A.16.2
  - Wide_Wide_Environment_Variables — A.17.1
  - Wide_Wide_Text_IO — A.11
    - Complex_IO — G.1.5
    - Editing — F.3.5
    - Text_Streams — A.12.4
    - Wide_Wide_Bounded_IO — A.11
    - Wide_Wide_Unbounded_IO — A.11

Interfaces — B.2
- C — B.3
  - Pointers — B.3.2
  - Strings — B.3.1
- COBOL — B.4
- Fortran — B.5

System — 13.7
- Address_To_Access_Conversions — 13.7.2
- Atomic_Operations — C.6.1
  - Exchange — C.6.2
  - Integer_Arithmetic — C.6.4
  - Modular_Arithmetic — C.6.5
  - Test_And_Set — C.6.3
- Machine_Code — 13.8
- Multiprocessors — D.16
  - Dispatching_Domains — D.16.1
- RPC — E.5
- Storage_Elements — 13.7.1
- Storage_Pools — 13.11
  - Subpools — 13.11.4]

2.a **Discussion:** In running text, we generally leave out the “Ada.” when referring to a child of Ada.

2.b **Reason:** We had no strict rule for which of Ada, Interfaces, or System should be the parent of a given library unit. However, we have tried to place as many things as possible under Ada, except that interfacing is a separate category, and we have tried to place library units whose use is highly nonportable under System.

Implementation Requirements

3/5 {*AI95-00434-01*} {*AI12-0052-1*} {*AI12-0114-1*} {*AI12-0200-1*} {*AI12-0439-1*} The implementation shall ensure that concurrent calls on any two (possibly the same) language-defined subprograms perform as specified, so long as all pairs of objects (one from each call) that are either denoted by parameters that can be passed by reference, or are designated by parameters of an access type, are nonoverlapping.

3.a.1/4 **Ramification:** {*AI12-0052-1*} {*AI12-0114-1*} So long as the parameters are disjoint, concurrent calls on the same language-defined subprogram, and concurrent calls on two different language-defined subprograms are required to work. But concurrent calls operating on overlapping objects (be they of the same or different language-defined subprograms) are *not* required to work (being an erroneous use of shared variables) unless both subprograms are required to pass the associated parameter by-copy.

3.a For example, simultaneous calls to Text_IO.Put will work properly, so long as they are going to two different files. On the other hand, simultaneous output to the same file constitutes erroneous use of shared variables.

3.b **To be honest:** Here, “language defined subprogram” means a language defined library subprogram, a subprogram declared in the visible part of a language defined library package, an instance of a language defined generic library subprogram, or a subprogram declared in the visible part of an instance of a language defined generic library package.

3.b.1/4 **Ramification:** {*AI12-0052-1*} This rule applies to all language-defined subprograms, including those defined in packages that manage some global state (like environment variables or the current directory). Unless specified above, such subprograms need to work when the explicit parameters are not overlapping; in particular, the existence of the global state is not considered.

3.c/4 {*AI12-0052-1*} The rule implies that any data local to the private part or body of the package (including global state as described above) has to be somehow protected against simultaneous access.

3.1/4 {*AI12-0052-1*} {*AI12-0159-1*} For the purpose of determining whether concurrent calls on text input-output subprograms are required to perform as specified above, when calling a subprogram within Text_IO or its children that implicitly operates on one of the default input-output files, the subprogram is considered to have a parameter of Current_Input or Current_Output (as appropriate).

{*AI05-0048-1*} If a descendant of a language-defined tagged type is declared, the implementation shall ensure that each inherited language-defined subprogram behaves as described in this Reference Manual. In particular, overriding a language-defined subprogram shall not alter the effect of any inherited language-defined subprogram. 3.2/3

Reason: This means that internally the implementation must not do redispatching unless it is required by the Standard. So when we say that some subprogram Bar is equivalent to Foo, overriding Foo for a derived type doesn't change the semantics of Bar, and in particular it means that Bar may no longer be equivalent to Foo. The word “equivalent” is always a bit of a lie anyway. 3.d/3

Implementation Permissions

The implementation may restrict the replacement of language-defined compilation units. The implementation may restrict children of language-defined library units (other than Standard). 4

Ramification: For example, the implementation may say, “you cannot compile a library unit called System” or “you cannot compile a child of package System” or “if you compile a library unit called System, it has to be a package, and it has to contain at least the following declarations: ...”. 4.a

Ramification: {*AI12-0112-1*} Implementations are of course allowed to make changes to the specifications of language-defined units, so long as those changes are semantically neutral (that is, no program could change legality or effect because of the changes). In particular, an implementation can **with** additional units (especially implementation-defined units) so long as those units do not change the elaboration of the language-defined unit. 4.b/5

Similarly, an implementation can add postconditions to language-defined subprograms, so long as those postconditions always evaluate to True. This is useful if the implementation can use those postconditions for optimization. 4.c/5

Wording Changes from Ada 83

Many of Ada 83's language-defined library units are now children of Ada or System. For upward compatibility, these are renamed as root library units (see J.1). 4.d

The order and lettering of the annexes has been changed. 4.e

Wording Changes from Ada 95

{*8652/0047*} {*AI95-00081-01*} **Corrigendum:** Units missing from the list of predefined units were added. 4.f/2

{*AI95-00424-01*} Added new units to the list of predefined units. 4.g/2

Wording Changes from Ada 2005

{*AI05-0048-1*} **Correction:** Added wording to ban redispatching unless it is explicitly required, in order to safeguard portability when overriding language-defined routines. 4.h/3

{*AI05-0060-1*} {*AI05-0206-1*} **Correction:** Added a permission to omit pragma Remote_Types from language-defined units if Annex E is not supported. This was later removed, as a better method of supporting the reason is now available. Note that this requires all implementations to provide minimal support for the Remote_Types categorization even if Annex E is not supported; being unable to compile language-defined units is not allowed. 4.i/3

{*AI05-0001-1*} {*AI05-0049-1*} {*AI05-0069-1*} {*AI05-0111-3*} {*AI05-0136-1*} {*AI05-0137-1*} {*AI05-0166-1*} {*AI05-0168-1*} Added various new units to the list of predefined units. 4.j/3

Incompatibilities With Ada 2012

{*AI12-0005-1*} When a new entity *E* is added to a package *P* that is **use**d in client code, *use clause conflicts* are possible. Specifically, if *P* is referenced in a use_clause, and an entity *F* with the same defining identifier as *E* is defined in some other package that is also referenced in a *use_clause*, the user-defined entity *F* may no longer be use-visible, resulting in errors. This is an incompatibility when the new entity is added to a language-defined package. Note that use clause conflicts are rare and easily fixed by using an expanded name. 4.k/5

Wording Changes from Ada 2012

{*AI12-0052-1*} {*AI12-0114-1*} {*AI12-0159-1*} **Corrigendum:** The rules requiring concurrent access of language-defined subprograms were expanded to include implicit Text_IO objects, overlapping objects designated by parameters of an access type, and simultaneous calls on different language-defined subprograms. While this might change behavior of some programs, it would do so by eliminating erroneous execution, so we don't consider this an inconsistency. 4.k.1/4

4.k.2/5 {*AI12-0200-1*} **Correction:** The rules requiring concurrent access of language-defined subprograms were clarified further.

4.l/5 {*AI12-0021-1*} {*AI12-0208-1*} {*AI12-0234-1*} {*AI12-0254-1*} {*AI12-0293-1*} {*AI12-0321-1*} {*AI12-0340-1*} Added various new units to the list of predefined units.

A.1 The Package Standard

1/3 {*AI05-0299-1*} This subclause outlines the specification of the package Standard containing all predefined identifiers in the language. The corresponding package body is not specified by the language.

2 The operators that are predefined for the types declared in the package Standard are given in comments since they are implicitly declared. Italics are used for pseudo-names of anonymous types (such as *root_real*) and for undefined information (such as *implementation-defined*).

2.a **Ramification:** All of the predefined operators are of convention Intrinsic.

Static Semantics

3 The library package Standard has the following declaration:

3.a **Implementation defined:** The names and characteristics of the numeric subtypes declared in the visible part of package Standard.

4/5
```
{AI12-0414-1} package Standard
   with Pure is
```

5
```
   type Boolean is (False, True);
```

6
```
   -- The predefined relational operators for this type are as follows:
```

7/1
```
{8652/0028} {AI95-00145-01}    -- function "="   (Left, Right : Boolean'Base)
return Boolean;
   -- function "/="  (Left, Right : Boolean'Base) return Boolean;
   -- function "<"   (Left, Right : Boolean'Base) return Boolean;
   -- function "<="  (Left, Right : Boolean'Base) return Boolean;
   -- function ">"   (Left, Right : Boolean'Base) return Boolean;
   -- function ">="  (Left, Right : Boolean'Base) return Boolean;
```

8
```
   -- The predefined logical operators and the predefined logical
   -- negation operator are as follows:
```

9/1
```
{8652/0028} {AI95-00145-01}    -- function "and" (Left, Right : Boolean'Base)
return Boolean'Base;
   -- function "or"  (Left, Right : Boolean'Base) return Boolean'Base;
   -- function "xor" (Left, Right : Boolean'Base) return Boolean'Base;
```

10/1
```
{8652/0028} {AI95-00145-01}    -- function "not" (Right : Boolean'Base) return
Boolean'Base;
```

11/2
```
{AI95-00434-01}     -- The integer type root_integer and the
   -- corresponding universal type universal_integer are predefined.
```

12
```
   type Integer is range implementation-defined;
```

13
```
   subtype Natural  is Integer range 0 .. Integer'Last;
   subtype Positive is Integer range 1 .. Integer'Last;
```

14
```
   -- The predefined operators for type Integer are as follows:
```

15
```
   -- function "="  (Left, Right : Integer'Base) return Boolean;
   -- function "/=" (Left, Right : Integer'Base) return Boolean;
   -- function "<"  (Left, Right : Integer'Base) return Boolean;
   -- function "<=" (Left, Right : Integer'Base) return Boolean;
   -- function ">"  (Left, Right : Integer'Base) return Boolean;
   -- function ">=" (Left, Right : Integer'Base) return Boolean;
```

16
```
   -- function "+"   (Right : Integer'Base) return Integer'Base;
   -- function "-"   (Right : Integer'Base) return Integer'Base;
   -- function "abs" (Right : Integer'Base) return Integer'Base;
```

```
   -- function "+"   (Left, Right : Integer'Base) return Integer'Base;                17
   -- function "-"   (Left, Right : Integer'Base) return Integer'Base;
   -- function "*"   (Left, Right : Integer'Base) return Integer'Base;
   -- function "/"   (Left, Right : Integer'Base) return Integer'Base;
   -- function "rem" (Left, Right : Integer'Base) return Integer'Base;
   -- function "mod" (Left, Right : Integer'Base) return Integer'Base;

   -- function "**"  (Left : Integer'Base; Right : Natural)                           18
   --                  return Integer'Base;

   -- The specification of each operator for the type                                 19
   -- root_integer, or for any additional predefined integer
   -- type, is obtained by replacing Integer by the name of the type
   -- in the specification of the corresponding operator of the type
   -- Integer. The right operand of the exponentiation operator
   -- remains as subtype Natural.

{AI95-00434-01}    -- The floating point type root_real and the                       20/2
   -- corresponding universal type universal_real are predefined.

   type Float is digits implementation-defined;                                       21

   -- The predefined operators for this type are as follows:                          22

   -- function "="   (Left, Right : Float) return Boolean;                           23
   -- function "/="  (Left, Right : Float) return Boolean;
   -- function "<"   (Left, Right : Float) return Boolean;
   -- function "<="  (Left, Right : Float) return Boolean;
   -- function ">"   (Left, Right : Float) return Boolean;
   -- function ">="  (Left, Right : Float) return Boolean;

   -- function "+"   (Right : Float) return Float;                                    24
   -- function "-"   (Right : Float) return Float;
   -- function "abs" (Right : Float) return Float;

   -- function "+"   (Left, Right : Float) return Float;                              25
   -- function "-"   (Left, Right : Float) return Float;
   -- function "*"   (Left, Right : Float) return Float;
   -- function "/"   (Left, Right : Float) return Float;

   -- function "**"  (Left : Float; Right : Integer'Base) return Float;               26

   -- The specification of each operator for the type root_real, or for               27
   -- any additional predefined floating point type, is obtained by
   -- replacing Float by the name of the type in the specification of the
   -- corresponding operator of the type Float.

   -- In addition, the following operators are predefined for the root                28
   -- numeric types:

   function "*" (Left : root_integer; Right : root_real)                              29
     return root_real;

   function "*" (Left : root_real;    Right : root_integer)                           30
     return root_real;

   function "/" (Left : root_real;    Right : root_integer)                           31
     return root_real;

   -- The type universal_fixed is predefined.                                         32
   -- The only multiplying operators defined between
   -- fixed point types are

   function "*" (Left : universal_fixed; Right : universal_fixed)                     33
     return universal_fixed;

   function "/" (Left : universal_fixed; Right : universal_fixed)                     34
     return universal_fixed;

{AI95-00230-01}    -- The type universal_access is predefined.                       34.1/2
   -- The following equality operators are predefined:

{AI95-00230-01}    function "=" (Left, Right: universal_access) return Boolean;      34.2/2
   function "/=" (Left, Right: universal_access) return Boolean;
```

35/3 {AI95-00415-01} {AI05-0181-1} {AI05-0248-1} -- The declaration of type Character is based on the standard ISO 8859-1 character set.

```
   -- There are no character literals corresponding to the positions for control characters.
   -- They are indicated in italics in this definition. See 3.5.2.

type Character is
  (nul,    soh,    stx,    etx,      eot,    enq,    ack,    bel,    --0 (16#00#) .. 7 (16#07#)
   bs,     ht,     lf,     vt,       ff,     cr,     so,     si,     --8 (16#08#) .. 15 (16#0F#)

   dle,    dc1,    dc2,    dc3,      dc4,    nak,    syn,    etb,    --16 (16#10#) .. 23 (16#17#)
   can,    em,     sub,    esc,      fs,     gs,     rs,     us,     --24 (16#18#) .. 31 (16#1F#)

   ' ',    '!',    '"',    '#',      '$',    '%',    '&',    ''',    --32 (16#20#) .. 39 (16#27#)
   '(',    ')',    '*',    '+',      ',',    '-',    '.',    '/',    --40 (16#28#) .. 47 (16#2F#)

   '0',    '1',    '2',    '3',      '4',    '5',    '6',    '7',    --48 (16#30#) .. 55 (16#37#)
   '8',    '9',    ':',    ';',      '<',    '=',    '>',    '?',    --56 (16#38#) .. 63 (16#3F#)

   '@',    'A',    'B',    'C',      'D',    'E',    'F',    'G',    --64 (16#40#) .. 71 (16#47#)
   'H',    'I',    'J',    'K',      'L',    'M',    'N',    'O',    --72 (16#48#) .. 79 (16#4F#)

   'P',    'Q',    'R',    'S',      'T',    'U',    'V',    'W',    --80 (16#50#) .. 87 (16#57#)
   'X',    'Y',    'Z',    '[',      '\',    ']',    '^',    '_',    --88 (16#58#) .. 95 (16#5F#)

   '`',    'a',    'b',    'c',      'd',    'e',    'f',    'g',    --96 (16#60#) .. 103 (16#67#)
   'h',    'i',    'j',    'k',      'l',    'm',    'n',    'o',    --104 (16#68#) .. 111 (16#6F#)

   'p',    'q',    'r',    's',      't',    'u',    'v',    'w',    --112 (16#70#) .. 119 (16#77#)
   'x',    'y',    'z',    '{',      '|',    '}',    '~',    del,    --120 (16#78#) .. 127 (16#7F#)

   reserved_128,  reserved_129,  bph,    nbh,                      --128 (16#80#) .. 131 (16#83#)
   reserved_132,  nel,    ssa,   esa,                              --132 (16#84#) .. 135 (16#87#)
   hts,    htj,    vts,    pld,      plu,    ri,     ss2,    ss3,    --136 (16#88#) .. 143 (16#8F#)

   dcs,    pu1,    pu2,    sts,      cch,    mw,     spa,    epa,    --144 (16#90#) .. 151 (16#97#)
   sos,    reserved_153,  sci,    csi,                              --152 (16#98#) .. 155 (16#9B#)
   st,     osc,    pm,     apc,                                     --156 (16#9C#) .. 159 (16#9F#)

   ' ',    '¡',    '¢',    '£',      '¤',    '¥',    '¦',    '§',    --160 (16#A0#) .. 167 (16#A7#)
   '¨',    '©',    'ª',    '«',                                     --168 (16#A8#) .. 171 (16#AB#)
   '¬',    soft_hyphen,   '®',    '¯',                             --172 (16#AC#) .. 175 (16#AF#)

   '°',    '±',    '²',    '³',      '´',    'µ',    '¶',    '·',    --176 (16#B0#) .. 183 (16#B7#)
   '¸',    '¹',    'º',    '»',      '¼',    '½',    '¾',    '¿',    --184 (16#B8#) .. 191 (16#BF#)

   'À',    'Á',    'Â',    'Ã',      'Ä',    'Å',    'Æ',    'Ç',    --192 (16#C0#) .. 199 (16#C7#)
   'È',    'É',    'Ê',    'Ë',      'Ì',    'Í',    'Î',    'Ï',    --200 (16#C8#) .. 207 (16#CF#)

   'Ð',    'Ñ',    'Ò',    'Ó',      'Ô',    'Õ',    'Ö',    '×',    --208 (16#D0#) .. 215 (16#D7#)
   'Ø',    'Ù',    'Ú',    'Û',      'Ü',    'Ý',    'Þ',    'ß',    --216 (16#D8#) .. 223 (16#DF#)

   'à',    'á',    'â',    'ã',      'ä',    'å',    'æ',    'ç',    --224 (16#E0#) .. 231 (16#E7#)
   'è',    'é',    'ê',    'ë',      'ì',    'í',    'î',    'ï',    --232 (16#E8#) .. 239 (16#EF#)

   'ð',    'ñ',    'ò',    'ó',      'ô',    'õ',    'ö',    '÷',    --240 (16#F0#) .. 247 (16#F7#)
   'ø',    'ù',    'ú',    'û',      'ü',    'ý',    'þ',    'ÿ');--248 (16#F8#) .. 255 (16#FF#)
```

36 -- The predefined operators for the type Character are the same as for
-- any enumeration type.

36.1/5 {AI95-00395-01} {AI05-0266-1} {AI12-0263-1} {AI12-0450-1} -- The declaration of type Wide_Character is based on the standard ISO/IEC 10646:2020 BMP character
-- set. The first 256 positions have the same contents as type Character. See 3.5.2.

```
type Wide_Character is (nul, soh ... Hex_0000FFFE, Hex_0000FFFF);
```

```
{AI95-00285-01} {AI95-00395-01} {AI05-0266-1} {AI12-0263-1} {AI12-0450-1}    -- The declaration of type Wide_Wide_Character is based on the full
   -- ISO/IEC 10646:2020 character set. The first 65536 positions have the
   -- same contents as type Wide_Character. See 3.5.2.

   type Wide_Wide_Character is (nul, soh ... Hex_7FFFFFFE, Hex_7FFFFFFF);
   for Wide_Wide_Character'Size use 32;

   package ASCII is ... end ASCII;  --Obsolescent; see J.5
```

36.2/5

36.3/2

```
{AI05-0229-1}    -- Predefined string types:
```

37/3

```
   type String is array(Positive range <>) of Character
      with Pack;

   -- The predefined operators for this type are as follows:
```

38

```
   --     function "="  (Left, Right: String) return Boolean;
   --     function "/=" (Left, Right: String) return Boolean;
   --     function "<"  (Left, Right: String) return Boolean;
   --     function "<=" (Left, Right: String) return Boolean;
   --     function ">"  (Left, Right: String) return Boolean;
   --     function ">=" (Left, Right: String) return Boolean;
```

39

```
   --     function "&" (Left: String;    Right: String)    return String;
   --     function "&" (Left: Character; Right: String)    return String;
   --     function "&" (Left: String;    Right: Character) return String;
   --     function "&" (Left: Character; Right: Character) return String;
```

40

```
{AI05-0229-1}    type Wide_String is array(Positive range <>) of Wide_Character
      with Pack;

   -- The predefined operators for this type correspond to those for String.
```

41/3

42

```
{AI95-00285-01} {AI05-0229-1}    type Wide_Wide_String is array (Positive range <>)
      of Wide_Wide_Character
         with Pack;

{AI95-00285-01}    -- The predefined operators for this type correspond to those for String.
```

42.1/3

42.2/2

```
   type Duration is delta implementation-defined range implementation-defined;
```

43

```
      -- The predefined operators for the type Duration are the same as for
      -- any fixed point type.
```

44

```
   -- The predefined exceptions:
```

45

```
   Constraint_Error: exception;
   Program_Error   : exception;
   Storage_Error   : exception;
   Tasking_Error   : exception;

end Standard;
```

46

47

Standard has no private part.

48

Reason: This is important for portability. All library packages are children of Standard, and if Standard had a private part then it would be visible to all of them.

48.a

{*AI95-00285-01*} In each of the types Character, Wide_Character, and Wide_Wide_Character, the character literals for the space character (position 32) and the non-breaking space character (position 160) correspond to different values. Unless indicated otherwise, each occurrence of the character literal ' ' in this Reference Manual refers to the space character. Similarly, the character literals for hyphen (position 45) and soft hyphen (position 173) correspond to different values. Unless indicated otherwise, each occurrence of the character literal '–' in this Reference Manual refers to the hyphen character.

49/2

Dynamic Semantics

Elaboration of the body of Standard has no effect.

50

50.a **Discussion:** Note that the language does not define where this body appears in the environment declarative_part — see 10, “Program Structure and Compilation Issues”.

Implementation Permissions

51 An implementation may provide additional predefined integer types and additional predefined floating point types. Some or all of these types may be anonymous.

51.a **To be honest:** An implementation may add representation items to package Standard, for example to specify the internal codes of type Boolean, or the Small of type Duration.

Implementation Advice

52 If an implementation provides additional named predefined integer types, then the names should end with “Integer” as in “Long_Integer”. If an implementation provides additional named predefined floating point types, then the names should end with “Float” as in “Long_Float”.

52.a/2 **Implementation Advice:** If an implementation provides additional named predefined integer types, then the names should end with “Integer”. If an implementation provides additional named predefined floating point types, then the names should end with “Float”.

53 NOTE 1 Certain aspects of the predefined entities cannot be completely described in the language itself. For example, although the enumeration type Boolean can be written showing the two enumeration literals False and True, the short-circuit control forms cannot be expressed in the language.

54 NOTE 2 As explained in 8.1, “Declarative Region” and 10.1.4, “The Compilation Process”, the declarative region of the package Standard encloses every library unit and consequently the main subprogram; the declaration of every library unit is assumed to occur within this declarative region. Library_items are assumed to be ordered in such a way that there are no forward semantic dependences. However, as explained in 8.3, “Visibility”, the only library units that are visible within a given compilation unit are the library units named by all with_clauses that apply to the given unit, and moreover, within the declarative region of a given library unit, that library unit itself.

55 NOTE 3 If all block_statements of a program are named, then the name of each program unit can always be written as an expanded name starting with Standard (unless Standard is itself hidden). The name of a library unit cannot be a homograph of a name (such as Integer) that is already declared in Standard.

56 NOTE 4 The exception Standard.Numeric_Error is defined in J.6.

56.a **Discussion:** The declaration of Natural needs to appear between the declaration of Integer and the (implicit) declaration of the "**" operator for Integer, because a formal parameter of "**" is of subtype Natural. This would be impossible in normal code, because the implicit declarations for a type occur immediately after the type declaration, with no possibility of intervening explicit declarations. But we're in Standard, and Standard is somewhat magic anyway.

56.b Using Natural as the subtype of the formal of "**" seems natural; it would be silly to have a textual rule about Constraint_Error being raised when there is a perfectly good subtype that means just that. Furthermore, by not using Integer for that formal, it helps remind the reader that the exponent remains Natural even when the left operand is replaced with the derivative of Integer. It doesn't logically imply that, but it's still useful as a reminder.

56.c In any case, declaring these general-purpose subtypes of Integer close to Integer seems more readable than declaring them much later.

Extensions to Ada 83

56.d Package Standard is declared to be pure.

56.e **Discussion:** The introduction of the types Wide_Character and Wide_String is not an Ada 95 extension to Ada 83, since ISO WG9 has approved these as an authorized extension of the original Ada 83 standard that is part of that standard.

Wording Changes from Ada 83

56.f Numeric_Error is made obsolescent.

56.g The declarations of Natural and Positive are moved to just after the declaration of Integer, so that "**" can refer to Natural without a forward reference. There's no real need to move Positive, too — it just came along for the ride.

Extensions to Ada 95

56.h/2 {*AI95-00285-01*} Types Wide_Wide_Character and Wide_Wide_String are new.

Discussion: The inconsistencies associated with these types are documented in 3.5.2 and 3.6.3. 56.i/2

{*AI95-00230-01*} Type *universal_access* and the equality operations for it are new. 56.j/2

Wording Changes from Ada 95

{*8652/0028*} {*AI95-00145-01*} **Corrigendum:** Corrected the parameter type for the Boolean operators declared in Standard. 56.k/2

Wording Changes from Ada 2005

{*AI05-0181-1*} **Correction:** Since soft_hyphen (position 173) is defined to be nongraphic, gave it a name. 56.l/3

Discussion: The inconsistencies associated with this change are documented in 3.5. 56.m/3

A.2 The Package Ada

Static Semantics

The following language-defined library package exists: 1

```
{AI12-0414-1} package Ada                                    2/5
   with Pure is
end Ada;
```

{*AI12-0414-1*} Ada serves as the parent of most of the other language-defined library units; its declaration is empty. 3/5

Legality Rules

In the standard mode, it is illegal to compile a child of package Ada. 4

Reason: The intention is that mentioning, say, Ada.Text_IO in a with_clause is guaranteed (at least in the standard mode) to refer to the standard version of Ada.Text_IO. The user can compile a root library unit Text_IO that has no relation to the standard version of Text_IO. 4.a

Ramification: Note that Ada can have non-language-defined grandchildren, assuming the implementation allows it. Also, packages System and Interfaces can have children, assuming the implementation allows it. 4.b

Implementation Note: An implementation will typically support a nonstandard mode in which compiling the language defined library units is allowed. Whether or not this mode is made available to users is up to the implementer. 4.c

An implementation could theoretically have private children of Ada, since that would be semantically neutral. However, a programmer cannot compile such a library unit. 4.d

Extensions to Ada 83

{*AI05-0299-1*} This subclause is new to Ada 95. 4.e/3

A.3 Character Handling

{*AI95-00285-01*} {*AI05-0243-1*} {*AI05-0299-1*} This subclause presents the packages related to character processing: an empty declared pure package Characters and child packages Characters.Handling and Characters.Latin_1. The package Characters.Handling provides classification and conversion functions for Character data, and some simple functions for dealing with Wide_Character and Wide_Wide_Character data. The child package Characters.Latin_1 declares a set of constants initialized to values of type Character. 1/3

Extensions to Ada 83

{*AI05-0299-1*} This subclause is new to Ada 95. 1.a/3

Wording Changes from Ada 95

{*AI95-00285-01*} Included Wide_Wide_Character in this description; the individual changes are documented as extensions as needed. 1.b/2

A.3.1 The Packages Characters, Wide_Characters, and Wide_Wide_Characters

Static Semantics

1 The library package Characters has the following declaration:

2/5
```
{AI12-0414-1} package Ada.Characters
   with Pure is
end Ada.Characters;
```

3/2 {*AI95-00395-01*} The library package Wide_Characters has the following declaration:

4/5
```
{AI12-0414-1} package Ada.Wide_Characters
   with Pure is
end Ada.Wide_Characters;
```

5/2 {*AI95-00395-01*} The library package Wide_Wide_Characters has the following declaration:

6/5
```
{AI12-0414-1} package Ada.Wide_Wide_Characters
   with Pure is
end Ada.Wide_Characters;
```

Implementation Advice

7/3 {*AI95-00395-01*} {*AI05-0185-1*} If an implementation chooses to provide implementation-defined operations on Wide_Character or Wide_String (such as collating and sorting, etc.) it should do so by providing child units of Wide_Characters. Similarly if it chooses to provide implementation-defined operations on Wide_Wide_Character or Wide_Wide_String it should do so by providing child units of Wide_Wide_Characters.

7.a/2 **Implementation Advice:** Implementation-defined operations on Wide_Character, Wide_String, Wide_Wide_Character, and Wide_Wide_String should be child units of Wide_Characters or Wide_Wide_Characters.

Extensions to Ada 95

7.b/2 {*AI95-00395-01*} The packages Wide_Characters and Wide_Wide_Characters are new.

A.3.2 The Package Characters.Handling

Static Semantics

1 The library package Characters.Handling has the following declaration:

2/5
```
{AI95-00362-01} {AI95-00395-01} {AI12-0414-1} with Ada.Characters.Conversions;
package Ada.Characters.Handling
  with Pure is
```

```
--Character classification functions                                                    3
{AI05-0185-1} {AI12-0004-1}   function Is_Control           (Item : in Character)      4/5
return Boolean;
  function Is_Graphic           (Item : in Character) return Boolean;
  function Is_Letter            (Item : in Character) return Boolean;
  function Is_Lower             (Item : in Character) return Boolean;
  function Is_Upper             (Item : in Character) return Boolean;
  function Is_Basic             (Item : in Character) return Boolean;
  function Is_Digit             (Item : in Character) return Boolean;
  function Is_Decimal_Digit     (Item : in Character) return Boolean
                    renames Is_Digit;
  function Is_Hexadecimal_Digit (Item : in Character) return Boolean;
  function Is_Alphanumeric      (Item : in Character) return Boolean;
  function Is_Special           (Item : in Character) return Boolean;
  function Is_Line_Terminator   (Item : in Character) return Boolean;
  function Is_Mark              (Item : in Character) return Boolean;
  function Is_Other_Format      (Item : in Character) return Boolean;
  function Is_Punctuation_Connector (Item : in Character) return Boolean;
  function Is_Space             (Item : in Character) return Boolean;
  function Is_NFKC              (Item : in Character) return Boolean;

--Conversion functions for Character and String                                         5

  function To_Lower (Item : in Character) return Character;                           6
  function To_Upper (Item : in Character) return Character;
  function To_Basic (Item : in Character) return Character;

  function To_Lower (Item : in String) return String;                                 7
  function To_Upper (Item : in String) return String;
  function To_Basic (Item : in String) return String;

--Classifications of and conversions between Character and ISO 646                     8

  subtype ISO_646 is                                                                  9
    Character range Character'Val(0) .. Character'Val(127);

  function Is_ISO_646 (Item : in Character) return Boolean;                           10
  function Is_ISO_646 (Item : in String)    return Boolean;

  function To_ISO_646 (Item       : in Character;                                     11
                       Substitute : in ISO_646 := ' ')
    return ISO_646;

  function To_ISO_646 (Item       : in String;                                        12
                       Substitute : in ISO_646 := ' ')
    return String;

{AI95-00285-01} {AI95-00395-01} -- The functions Is_Character, Is_String, To_Character, To_String,   13/2
To_Wide_Character,
-- and To_Wide_String are obsolescent; see J.14.

Paragraphs 14 through 18 were deleted.

end Ada.Characters.Handling;                                                          19
```

Discussion: {*AI95-00395-01*} The with_clause for Ada.Characters.Conversions is needed for the definition of the obsolescent functions (see J.14). It would be odd to put this clause into J.14 as it was not present in Ada 95, and with_clauses are semantically neutral to clients anyway. 19.a/2

In the description below for each function that returns a Boolean result, the effect is described in terms of the conditions under which the value True is returned. If these conditions are not met, then the function returns False. 20

Each of the following classification functions has a formal Character parameter, Item, and returns a Boolean result. 21

Is_Control 22

True if Item is a control character. A *control character* is a character whose position is in one of the ranges 0..31 or 127..159.

23 Is_Graphic
True if Item is a graphic character. A *graphic character* is a character whose position is in one of the ranges 32..126 or 160..255.

24 Is_Letter True if Item is a letter. A *letter* is a character that is in one of the ranges 'A'..'Z' or 'a'..'z', or whose position is in one of the ranges 192..214, 216..246, or 248..255.

25 Is_Lower True if Item is a lower-case letter. A *lower-case letter* is a character that is in the range 'a'..'z', or whose position is in one of the ranges 223..246 or 248..255.

26 Is_Upper True if Item is an upper-case letter. An *upper-case letter* is a character that is in the range 'A'..'Z' or whose position is in one of the ranges 192..214 or 216.. 222.

27 Is_Basic True if Item is a basic letter. A *basic letter* is a character that is in one of the ranges 'A'..'Z' and 'a'..'z', or that is one of the following: 'Æ', 'æ', 'Ð', 'ð', 'Þ', 'þ', or 'ß'.

28 Is_Digit True if Item is a decimal digit. A *decimal digit* is a character in the range '0'..'9'.

29 Is_Decimal_Digit
A renaming of Is_Digit.

30 Is_Hexadecimal_Digit
True if Item is a hexadecimal digit. A *hexadecimal digit* is a character that is either a decimal digit or that is in one of the ranges 'A' .. 'F' or 'a' .. 'f'.

31 Is_Alphanumeric
True if Item is an alphanumeric character. An *alphanumeric character* is a character that is either a letter or a decimal digit.

32 Is_Special
True if Item is a special graphic character. A *special graphic character* is a graphic character that is not alphanumeric.

32.1/3 {*AI05-0185-1*} Is_Line_Terminator
True if Item is a character with position 10 .. 13 (Line_Feed, Line_Tabulation, Form_Feed, Carriage_Return) or 133 (Next_Line).

32.2/3 {*AI05-0185-1*} Is_Mark
Never True (no value of type Character has categories Mark, Non-Spacing or Mark, Spacing Combining).

32.3/3 {*AI05-0185-1*} Is_Other_Format
True if Item is a character with position 173 (Soft_Hyphen).

32.4/3 {*AI05-0185-1*} Is_Punctuation_Connector
True if Item is a character with position 95 ('_', known as Low_Line or Underscore).

32.5/3 {*AI05-0185-1*} Is_Space
True if Item is a character with position 32 (' ') or 160 (No_Break_Space).

32.6/5 {*AI12-0004-1*} {*AI12-0263-1*} {*AI12-0439-1*} {*AI12-0450-1*} Is_NFKC
True if Item can be present in a string normalized to Normalization Form KC (as defined by Clause 22 of ISO/IEC 10646:2020); this includes all characters except those with positions 160, 168, 170, 175, 178, 179, 180, 181, 184, 185, 186, 188, 189, and 190.

33 Each of the names To_Lower, To_Upper, and To_Basic refers to two functions: one that converts from Character to Character, and the other that converts from String to String. The result of each Character-to-Character function is described below, in terms of the conversion applied to Item, its formal Character parameter. The result of each String-to-String conversion is obtained by applying to each element of the function's String parameter the corresponding Character-to-Character conversion; the result is the null String if the value of the formal parameter is the null String. The lower bound of the result String is 1.

To_Lower Returns the corresponding lower-case value for Item if Is_Upper(Item), and returns Item otherwise. 34

To_Upper Returns the corresponding upper-case value for Item if Is_Lower(Item) and Item has an upper-case form, and returns Item otherwise. The lower case letters 'ß' and 'ÿ' do not have upper case forms. 35

To_Basic Returns the letter corresponding to Item but with no diacritical mark, if Item is a letter but not a basic letter; returns Item otherwise. 36

The following set of functions test for membership in the ISO 646 character range, or convert between ISO 646 and Character. 37

Is_ISO_646 38

The function whose formal parameter, Item, is of type Character returns True if Item is in the subtype ISO_646.

Is_ISO_646 39

The function whose formal parameter, Item, is of type String returns True if Is_ISO_646(Item(I)) is True for each I in Item'Range.

To_ISO_646 40

The function whose first formal parameter, Item, is of type Character returns Item if Is_ISO_646(Item), and returns the Substitute ISO_646 character otherwise.

To_ISO_646 41

The function whose first formal parameter, Item, is of type String returns the String whose Range is 1..Item'Length and each of whose elements is given by To_ISO_646 of the corresponding element in Item.

Paragraphs 42 through 49 were deleted.

NOTE 1 A basic letter is a letter without a diacritical mark. 50

NOTE 2 Except for the hexadecimal digits, basic letters, and ISO_646 characters, the categories identified in the classification functions form a strict hierarchy: 51

— Control characters 52

— Graphic characters 53

— Alphanumeric characters 54

— Letters 55

— Upper-case letters 56

— Lower-case letters 57

— Decimal digits 58

— Special graphic characters 59

Ramification: Thus each Character value is either a control character or a graphic character but not both; each graphic character is either an alphanumeric or special graphic but not both; each alphanumeric is either a letter or decimal digit but not both; each letter is either upper case or lower case but not both. 59.a

NOTE 3 {*AI05-0114-1*} There are certain characters which are defined to be lower case letters by ISO 10646 and are therefore allowed in identifiers, but are not considered lower case letters by Ada.Characters.Handling. 60/3

Reason: This is to maintain runtime compatibility with the Ada 95 definitions of these functions. We don't list the exact characters involved because they're likely to change in future character set standards; the list for ISO 10646:2011 can be found in AI05-0114-1. 60.a/3

Ramification: No version of Characters.Handling is intended to do portable (Ada-version independent) manipulation of Ada identifiers. The classification given by Wide_Characters.Handling will be correct for the current implementation for Ada 2012 identifiers, but it might not be correct for a different implementation or version of Ada. 60.b/3

Extensions to Ada 95

{*AI95-00362-01*} Characters.Handling is now Pure, so it can be used in pure units. 60.c/2

Incompatibilities With Ada 2005

60.d/3 {*AI05-0185-1*} Added additional classification routines so that Characters.Handling has all of the routines available in Wide_Characters.Handling. If Characters.Handling is referenced in a use_clause, and an entity *E* with a defining_identifier that is the same as one of the new functions is defined in a package that is also referenced in a use_clause, the entity *E* may no longer be use-visible, resulting in errors. This should be rare and is easily fixed if it does occur.

Wording Changes from Ada 95

60.e/2 {*AI95-00285-01*} {*AI95-00395-01*} The conversion functions are made obsolescent; a more complete set is available in Characters.Conversions — see A.3.4.

60.f/3 {*AI95-00285-01*} {*AI05-0248-1*} We no longer talk about localized character sets; these are a nonstandard mode, which is none of our business.

Wording Changes from Ada 2005

60.g/3 {*AI05-0114-1*} **Correction:** Added a note to clarify that these functions don't have any relationship to the characters allowed in identifiers.

Incompatibilities With Ada 2012

60.h/5 {*AI12-0004-1*} Added an additional classification routine Is_NFKC. Therefore, a use clause conflict is possible; see the introduction of Annex A for more on this topic.

A.3.3 The Package Characters.Latin_1

1 The package Characters.Latin_1 declares constants for characters in ISO 8859-1.

1.a **Reason:** The constants for the ISO 646 characters could have been declared as renamings of objects declared in package ASCII, as opposed to explicit constants. The main reason for explicit constants was for consistency of style with the upper-half constants, and to avoid emphasizing the package ASCII.

Static Semantics

2 The library package Characters.Latin_1 has the following declaration:

3/5
```
{AI12-0414-1} package Ada.Characters.Latin_1
    with Pure is
```

4
```
   -- Control characters:
```

5
```
   NUL                    : constant Character := Character'Val(0);
   SOH                    : constant Character := Character'Val(1);
   STX                    : constant Character := Character'Val(2);
   ETX                    : constant Character := Character'Val(3);
   EOT                    : constant Character := Character'Val(4);
   ENQ                    : constant Character := Character'Val(5);
   ACK                    : constant Character := Character'Val(6);
   BEL                    : constant Character := Character'Val(7);
   BS                     : constant Character := Character'Val(8);
   HT                     : constant Character := Character'Val(9);
   LF                     : constant Character := Character'Val(10);
   VT                     : constant Character := Character'Val(11);
   FF                     : constant Character := Character'Val(12);
   CR                     : constant Character := Character'Val(13);
   SO                     : constant Character := Character'Val(14);
   SI                     : constant Character := Character'Val(15);
```

```
   DLE                    : constant Character := Character'Val(16);
   DC1                    : constant Character := Character'Val(17);
   DC2                    : constant Character := Character'Val(18);
   DC3                    : constant Character := Character'Val(19);
   DC4                    : constant Character := Character'Val(20);
   NAK                    : constant Character := Character'Val(21);
   SYN                    : constant Character := Character'Val(22);
   ETB                    : constant Character := Character'Val(23);
   CAN                    : constant Character := Character'Val(24);
   EM                     : constant Character := Character'Val(25);
   SUB                    : constant Character := Character'Val(26);
   ESC                    : constant Character := Character'Val(27);
   FS                     : constant Character := Character'Val(28);
   GS                     : constant Character := Character'Val(29);
   RS                     : constant Character := Character'Val(30);
   US                     : constant Character := Character'Val(31);

-- ISO 646 graphic characters:

   Space                  : constant Character := ' ';  -- Character'Val(32)
   Exclamation            : constant Character := '!';  -- Character'Val(33)
   Quotation              : constant Character := '"';  -- Character'Val(34)
   Number_Sign            : constant Character := '#';  -- Character'Val(35)
   Dollar_Sign            : constant Character := '$';  -- Character'Val(36)
   Percent_Sign           : constant Character := '%';  -- Character'Val(37)
   Ampersand              : constant Character := '&';  -- Character'Val(38)
   Apostrophe             : constant Character := ''';  -- Character'Val(39)
   Left_Parenthesis       : constant Character := '(';  -- Character'Val(40)
   Right_Parenthesis      : constant Character := ')';  -- Character'Val(41)
   Asterisk               : constant Character := '*';  -- Character'Val(42)
   Plus_Sign              : constant Character := '+';  -- Character'Val(43)
   Comma                  : constant Character := ',';  -- Character'Val(44)
   Hyphen                 : constant Character := '-';  -- Character'Val(45)
   Minus_Sign             : Character renames Hyphen;
   Full_Stop              : constant Character := '.';  -- Character'Val(46)
   Solidus                : constant Character := '/';  -- Character'Val(47)

   -- Decimal digits '0' though '9' are at positions 48 through 57

   Colon                  : constant Character := ':';  -- Character'Val(58)
   Semicolon              : constant Character := ';';  -- Character'Val(59)
   Less_Than_Sign         : constant Character := '<';  -- Character'Val(60)
   Equals_Sign            : constant Character := '=';  -- Character'Val(61)
   Greater_Than_Sign      : constant Character := '>';  -- Character'Val(62)
   Question               : constant Character := '?';  -- Character'Val(63)
   Commercial_At          : constant Character := '@';  -- Character'Val(64)

   -- Letters 'A' through 'Z' are at positions 65 through 90

   Left_Square_Bracket    : constant Character := '[';  -- Character'Val(91)
   Reverse_Solidus        : constant Character := '\';  -- Character'Val(92)
   Right_Square_Bracket   : constant Character := ']';  -- Character'Val(93)
   Circumflex             : constant Character := '^';  -- Character'Val(94)
   Low_Line               : constant Character := '_';  -- Character'Val(95)

   Grave                  : constant Character := '`';  -- Character'Val(96)
   LC_A                   : constant Character := 'a';  -- Character'Val(97)
   LC_B                   : constant Character := 'b';  -- Character'Val(98)
   LC_C                   : constant Character := 'c';  -- Character'Val(99)
   LC_D                   : constant Character := 'd';  -- Character'Val(100)
   LC_E                   : constant Character := 'e';  -- Character'Val(101)
   LC_F                   : constant Character := 'f';  -- Character'Val(102)
   LC_G                   : constant Character := 'g';  -- Character'Val(103)
   LC_H                   : constant Character := 'h';  -- Character'Val(104)
   LC_I                   : constant Character := 'i';  -- Character'Val(105)
   LC_J                   : constant Character := 'j';  -- Character'Val(106)
   LC_K                   : constant Character := 'k';  -- Character'Val(107)
   LC_L                   : constant Character := 'l';  -- Character'Val(108)
   LC_M                   : constant Character := 'm';  -- Character'Val(109)
   LC_N                   : constant Character := 'n';  -- Character'Val(110)
   LC_O                   : constant Character := 'o';  -- Character'Val(111)
```

```
14        LC_P                 : constant Character := 'p';  -- Character'Val(112)
          LC_Q                 : constant Character := 'q';  -- Character'Val(113)
          LC_R                 : constant Character := 'r';  -- Character'Val(114)
          LC_S                 : constant Character := 's';  -- Character'Val(115)
          LC_T                 : constant Character := 't';  -- Character'Val(116)
          LC_U                 : constant Character := 'u';  -- Character'Val(117)
          LC_V                 : constant Character := 'v';  -- Character'Val(118)
          LC_W                 : constant Character := 'w';  -- Character'Val(119)
          LC_X                 : constant Character := 'x';  -- Character'Val(120)
          LC_Y                 : constant Character := 'y';  -- Character'Val(121)
          LC_Z                 : constant Character := 'z';  -- Character'Val(122)
          Left_Curly_Bracket   : constant Character := '{';  -- Character'Val(123)
          Vertical_Line        : constant Character := '|';  -- Character'Val(124)
          Right_Curly_Bracket  : constant Character := '}';  -- Character'Val(125)
          Tilde                : constant Character := '~';  -- Character'Val(126)
          DEL                  : constant Character := Character'Val(127);
15        -- ISO 6429 control characters:
16        IS4                  : Character renames FS;
          IS3                  : Character renames GS;
          IS2                  : Character renames RS;
          IS1                  : Character renames US;
17        Reserved_128         : constant Character := Character'Val(128);
          Reserved_129         : constant Character := Character'Val(129);
          BPH                  : constant Character := Character'Val(130);
          NBH                  : constant Character := Character'Val(131);
          Reserved_132         : constant Character := Character'Val(132);
          NEL                  : constant Character := Character'Val(133);
          SSA                  : constant Character := Character'Val(134);
          ESA                  : constant Character := Character'Val(135);
          HTS                  : constant Character := Character'Val(136);
          HTJ                  : constant Character := Character'Val(137);
          VTS                  : constant Character := Character'Val(138);
          PLD                  : constant Character := Character'Val(139);
          PLU                  : constant Character := Character'Val(140);
          RI                   : constant Character := Character'Val(141);
          SS2                  : constant Character := Character'Val(142);
          SS3                  : constant Character := Character'Val(143);
18        DCS                  : constant Character := Character'Val(144);
          PU1                  : constant Character := Character'Val(145);
          PU2                  : constant Character := Character'Val(146);
          STS                  : constant Character := Character'Val(147);
          CCH                  : constant Character := Character'Val(148);
          MW                   : constant Character := Character'Val(149);
          SPA                  : constant Character := Character'Val(150);
          EPA                  : constant Character := Character'Val(151);
19        SOS                  : constant Character := Character'Val(152);
          Reserved_153         : constant Character := Character'Val(153);
          SCI                  : constant Character := Character'Val(154);
          CSI                  : constant Character := Character'Val(155);
          ST                   : constant Character := Character'Val(156);
          OSC                  : constant Character := Character'Val(157);
          PM                   : constant Character := Character'Val(158);
          APC                  : constant Character := Character'Val(159);
```

```
   -- Other graphic characters:                                                    20
   {AI05-0181-1} -- Character positions 160 (16#A0#) .. 175 (16#AF#):             21/3
      No_Break_Space               : constant Character := ' '; --Character'Val(160)
      NBSP                         : Character renames No_Break_Space;
      Inverted_Exclamation         : constant Character := '¡'; --Character'Val(161)
      Cent_Sign                    : constant Character := '¢'; --Character'Val(162)
      Pound_Sign                   : constant Character := '£'; --Character'Val(163)
      Currency_Sign                : constant Character := '¤'; --Character'Val(164)
      Yen_Sign                     : constant Character := '¥'; --Character'Val(165)
      Broken_Bar                   : constant Character := '¦'; --Character'Val(166)
      Section_Sign                 : constant Character := '§'; --Character'Val(167)
      Diaeresis                    : constant Character := '¨'; --Character'Val(168)
      Copyright_Sign               : constant Character := '©'; --Character'Val(169)
      Feminine_Ordinal_Indicator   : constant Character := 'ª'; --Character'Val(170)
      Left_Angle_Quotation         : constant Character := '«'; --Character'Val(171)
      Not_Sign                     : constant Character := '¬'; --Character'Val(172)
      Soft_Hyphen                  : constant Character := Character'Val(173);
      Registered_Trade_Mark_Sign   : constant Character := '®'; --Character'Val(174)
      Macron                       : constant Character := '¯'; --Character'Val(175)
   -- Character positions 176 (16#B0#) .. 191 (16#BF#):                           22
      Degree_Sign                  : constant Character := '°'; --Character'Val(176)
      Ring_Above                   : Character renames Degree_Sign;
      Plus_Minus_Sign              : constant Character := '±'; --Character'Val(177)
      Superscript_Two              : constant Character := '²'; --Character'Val(178)
      Superscript_Three            : constant Character := '³'; --Character'Val(179)
      Acute                        : constant Character := '´'; --Character'Val(180)
      Micro_Sign                   : constant Character := 'µ'; --Character'Val(181)
      Pilcrow_Sign                 : constant Character := '¶'; --Character'Val(182)
      Paragraph_Sign               : Character renames Pilcrow_Sign;
      Middle_Dot                   : constant Character := '·'; --Character'Val(183)
      Cedilla                      : constant Character := '¸'; --Character'Val(184)
      Superscript_One              : constant Character := '¹'; --Character'Val(185)
      Masculine_Ordinal_Indicator: constant Character := 'º'; --Character'Val(186)
      Right_Angle_Quotation        : constant Character := '»'; --Character'Val(187)
      Fraction_One_Quarter         : constant Character := '¼'; --Character'Val(188)
      Fraction_One_Half            : constant Character := '½'; --Character'Val(189)
      Fraction_Three_Quarters      : constant Character := '¾'; --Character'Val(190)
      Inverted_Question            : constant Character := '¿'; --Character'Val(191)
   -- Character positions 192 (16#C0#) .. 207 (16#CF#):                           23
      UC_A_Grave                   : constant Character := 'À'; --Character'Val(192)
      UC_A_Acute                   : constant Character := 'Á'; --Character'Val(193)
      UC_A_Circumflex              : constant Character := 'Â'; --Character'Val(194)
      UC_A_Tilde                   : constant Character := 'Ã'; --Character'Val(195)
      UC_A_Diaeresis               : constant Character := 'Ä'; --Character'Val(196)
      UC_A_Ring                    : constant Character := 'Å'; --Character'Val(197)
      UC_AE_Diphthong              : constant Character := 'Æ'; --Character'Val(198)
      UC_C_Cedilla                 : constant Character := 'Ç'; --Character'Val(199)
      UC_E_Grave                   : constant Character := 'È'; --Character'Val(200)
      UC_E_Acute                   : constant Character := 'É'; --Character'Val(201)
      UC_E_Circumflex              : constant Character := 'Ê'; --Character'Val(202)
      UC_E_Diaeresis               : constant Character := 'Ë'; --Character'Val(203)
      UC_I_Grave                   : constant Character := 'Ì'; --Character'Val(204)
      UC_I_Acute                   : constant Character := 'Í'; --Character'Val(205)
      UC_I_Circumflex              : constant Character := 'Î'; --Character'Val(206)
      UC_I_Diaeresis               : constant Character := 'Ï'; --Character'Val(207)
```

24
```
   -- Character positions 208 (16#D0#) .. 223 (16#DF#):
      UC_Icelandic_Eth           : constant Character := 'Ð'; --Character'Val(208)
      UC_N_Tilde                 : constant Character := 'Ñ'; --Character'Val(209)
      UC_O_Grave                 : constant Character := 'Ò'; --Character'Val(210)
      UC_O_Acute                 : constant Character := 'Ó'; --Character'Val(211)
      UC_O_Circumflex            : constant Character := 'Ô'; --Character'Val(212)
      UC_O_Tilde                 : constant Character := 'Õ'; --Character'Val(213)
      UC_O_Diaeresis             : constant Character := 'Ö'; --Character'Val(214)
      Multiplication_Sign        : constant Character := '×'; --Character'Val(215)
      UC_O_Oblique_Stroke        : constant Character := 'Ø'; --Character'Val(216)
      UC_U_Grave                 : constant Character := 'Ù'; --Character'Val(217)
      UC_U_Acute                 : constant Character := 'Ú'; --Character'Val(218)
      UC_U_Circumflex            : constant Character := 'Û'; --Character'Val(219)
      UC_U_Diaeresis             : constant Character := 'Ü'; --Character'Val(220)
      UC_Y_Acute                 : constant Character := 'Ý'; --Character'Val(221)
      UC_Icelandic_Thorn         : constant Character := 'Þ'; --Character'Val(222)
      LC_German_Sharp_S          : constant Character := 'ß'; --Character'Val(223)
```
25
```
   -- Character positions 224 (16#E0#) .. 239 (16#EF#):
      LC_A_Grave                 : constant Character := 'à'; --Character'Val(224)
      LC_A_Acute                 : constant Character := 'á'; --Character'Val(225)
      LC_A_Circumflex            : constant Character := 'â'; --Character'Val(226)
      LC_A_Tilde                 : constant Character := 'ã'; --Character'Val(227)
      LC_A_Diaeresis             : constant Character := 'ä'; --Character'Val(228)
      LC_A_Ring                  : constant Character := 'å'; --Character'Val(229)
      LC_AE_Diphthong            : constant Character := 'æ'; --Character'Val(230)
      LC_C_Cedilla               : constant Character := 'ç'; --Character'Val(231)
      LC_E_Grave                 : constant Character := 'è'; --Character'Val(232)
      LC_E_Acute                 : constant Character := 'é'; --Character'Val(233)
      LC_E_Circumflex            : constant Character := 'ê'; --Character'Val(234)
      LC_E_Diaeresis             : constant Character := 'ë'; --Character'Val(235)
      LC_I_Grave                 : constant Character := 'ì'; --Character'Val(236)
      LC_I_Acute                 : constant Character := 'í'; --Character'Val(237)
      LC_I_Circumflex            : constant Character := 'î'; --Character'Val(238)
      LC_I_Diaeresis             : constant Character := 'ï'; --Character'Val(239)
```
26
```
   -- Character positions 240 (16#F0#) .. 255 (16#FF#):
      LC_Icelandic_Eth           : constant Character := 'ð'; --Character'Val(240)
      LC_N_Tilde                 : constant Character := 'ñ'; --Character'Val(241)
      LC_O_Grave                 : constant Character := 'ò'; --Character'Val(242)
      LC_O_Acute                 : constant Character := 'ó'; --Character'Val(243)
      LC_O_Circumflex            : constant Character := 'ô'; --Character'Val(244)
      LC_O_Tilde                 : constant Character := 'õ'; --Character'Val(245)
      LC_O_Diaeresis             : constant Character := 'ö'; --Character'Val(246)
      Division_Sign              : constant Character := '÷'; --Character'Val(247)
      LC_O_Oblique_Stroke        : constant Character := 'ø'; --Character'Val(248)
      LC_U_Grave                 : constant Character := 'ù'; --Character'Val(249)
      LC_U_Acute                 : constant Character := 'ú'; --Character'Val(250)
      LC_U_Circumflex            : constant Character := 'û'; --Character'Val(251)
      LC_U_Diaeresis             : constant Character := 'ü'; --Character'Val(252)
      LC_Y_Acute                 : constant Character := 'ý'; --Character'Val(253)
      LC_Icelandic_Thorn         : constant Character := 'þ'; --Character'Val(254)
      LC_Y_Diaeresis             : constant Character := 'ÿ'; --Character'Val(255)
end Ada.Characters.Latin_1;
```

Implementation Permissions

27 An implementation may provide additional packages as children of Ada.Characters, to declare names for the symbols of the local character set or other character sets.

Wording Changes from Ada 2005

27.a/3 {*AI05-0181-1*} **Correction:** Soft_Hyphen is not a graphic character, and thus a character literal for it is illegal. So we have to use the position value. This makes no semantic change to users of the constant.

A.3.4 The Package Characters.Conversions

Static Semantics

{*AI95-00395-01*} The library package Characters.Conversions has the following declaration: 1/2

```
{AI12-0414-1} package Ada.Characters.Conversions                                    2/5
   with Pure is

   function Is_Character (Item : in Wide_Character)      return Boolean;            3/2
   function Is_String    (Item : in Wide_String)         return Boolean;
   function Is_Character (Item : in Wide_Wide_Character) return Boolean;
   function Is_String    (Item : in Wide_Wide_String)    return Boolean;
   function Is_Wide_Character (Item : in Wide_Wide_Character)
      return Boolean;
   function Is_Wide_String    (Item : in Wide_Wide_String)
      return Boolean;

   function To_Wide_Character (Item : in Character) return Wide_Character;        4/2
   function To_Wide_String    (Item : in String)    return Wide_String;
   function To_Wide_Wide_Character (Item : in Character)
      return Wide_Wide_Character;
   function To_Wide_Wide_String    (Item : in String)
      return Wide_Wide_String;
   function To_Wide_Wide_Character (Item : in Wide_Character)
      return Wide_Wide_Character;
   function To_Wide_Wide_String    (Item : in Wide_String)
      return Wide_Wide_String;

   function To_Character (Item       : in Wide_Character;                         5/2
                          Substitute : in Character := ' ')
      return Character;
   function To_String    (Item       : in Wide_String;
                          Substitute : in Character := ' ')
      return String;
   function To_Character (Item :       in Wide_Wide_Character;
                          Substitute : in Character := ' ')
      return Character;
   function To_String    (Item :       in Wide_Wide_String;
                          Substitute : in Character := ' ')
      return String;
   function To_Wide_Character (Item :       in Wide_Wide_Character;
                               Substitute : in Wide_Character := ' ')
      return Wide_Character;
   function To_Wide_String    (Item :       in Wide_Wide_String;
                               Substitute : in Wide_Character := ' ')
      return Wide_String;

end Ada.Characters.Conversions;                                                    6/2
```

{*AI95-00395-01*} The functions in package Characters.Conversions test Wide_Wide_Character or Wide_Character values for membership in Wide_Character or Character, or convert between corresponding characters of Wide_Wide_Character, Wide_Character, and Character. 7/2

```
function Is_Character (Item : in Wide_Character) return Boolean;                   8/2
```

{*AI95-00395-01*} Returns True if Wide_Character'Pos(Item) <= Character'Pos(Character'Last). 9/2

```
function Is_Character (Item : in Wide_Wide_Character) return Boolean;              10/2
```

{*AI95-00395-01*} Returns True if Wide_Wide_Character'Pos(Item) <= Character'Pos(Character'Last). 11/2

```
function Is_Wide_Character (Item : in Wide_Wide_Character) return Boolean;         12/2
```

{*AI95-00395-01*} Returns True if Wide_Wide_Character'Pos(Item) <= Wide_Character'Pos(Wide_Character'Last). 13/2

14/2
```
function Is_String (Item : in Wide_String)      return Boolean;
function Is_String (Item : in Wide_Wide_String) return Boolean;
```

15/2 {*AI95-00395-01*} Returns True if Is_Character(Item(I)) is True for each I in Item'Range.

16/2
```
function Is_Wide_String (Item : in Wide_Wide_String) return Boolean;
```

17/2 {*AI95-00395-01*} Returns True if Is_Wide_Character(Item(I)) is True for each I in Item'Range.

18/2
```
function To_Character (Item :       in Wide_Character;
                       Substitute : in Character := ' ') return Character;
function To_Character (Item :       in Wide_Wide_Character;
                       Substitute : in Character := ' ') return Character;
```

19/2 {*AI95-00395-01*} Returns the Character corresponding to Item if Is_Character(Item), and returns the Substitute Character otherwise.

20/2
```
function To_Wide_Character (Item : in Character) return Wide_Character;
```

21/2 {*AI95-00395-01*} Returns the Wide_Character X such that Character'Pos(Item) = Wide_Character'Pos (X).

22/2
```
function To_Wide_Character (Item :       in Wide_Wide_Character;
                            Substitute : in Wide_Character := ' ')
   return Wide_Character;
```

23/2 {*AI95-00395-01*} Returns the Wide_Character corresponding to Item if Is_Wide_Character(Item), and returns the Substitute Wide_Character otherwise.

24/2
```
function To_Wide_Wide_Character (Item : in Character)
   return Wide_Wide_Character;
```

25/2 {*AI95-00395-01*} Returns the Wide_Wide_Character X such that Character'Pos(Item) = Wide_Wide_Character'Pos (X).

26/2
```
function To_Wide_Wide_Character (Item : in Wide_Character)
   return Wide_Wide_Character;
```

27/2 {*AI95-00395-01*} Returns the Wide_Wide_Character X such that Wide_Character'Pos(Item) = Wide_Wide_Character'Pos (X).

28/2
```
function To_String (Item :       in Wide_String;
                    Substitute : in Character := ' ') return String;
function To_String (Item :       in Wide_Wide_String;
                    Substitute : in Character := ' ') return String;
```

29/2 {*AI95-00395-01*} Returns the String whose range is 1..Item'Length and each of whose elements is given by To_Character of the corresponding element in Item.

30/2
```
function To_Wide_String (Item : in String) return Wide_String;
```

31/2 {*AI95-00395-01*} Returns the Wide_String whose range is 1..Item'Length and each of whose elements is given by To_Wide_Character of the corresponding element in Item.

32/2
```
function To_Wide_String (Item :       in Wide_Wide_String;
                         Substitute : in Wide_Character := ' ')
   return Wide_String;
```

33/2 {*AI95-00395-01*} Returns the Wide_String whose range is 1..Item'Length and each of whose elements is given by To_Wide_Character of the corresponding element in Item with the given Substitute Wide_Character.

```
function To_Wide_Wide_String (Item : in String) return Wide_Wide_String;
function To_Wide_Wide_String (Item : in Wide_String)
   return Wide_Wide_String;
```

34/2

{*AI95-00395-01*} Returns the Wide_Wide_String whose range is 1..Item'Length and each of whose elements is given by To_Wide_Wide_Character of the corresponding element in Item.

35/2

Extensions to Ada 95

{*AI95-00395-01*} The package Characters.Conversions is new, replacing functions previously found in Characters.Handling.

35.a/2

A.3.5 The Package Wide_Characters.Handling

{*AI05-0185-1*} The package Wide_Characters.Handling provides operations for classifying Wide_Characters and case folding for Wide_Characters.

1/3

Static Semantics

{*AI05-0185-1*} The library package Wide_Characters.Handling has the following declaration:

2/3

```
{AI05-0185-1} {AI05-0266-1} {AI12-0414-1} package Ada.Wide_Characters.Handling
   with Pure is
```

3/5

```
{AI05-0266-1}    function Character_Set_Version return String;
```

4/3

```
   function Is_Control (Item : Wide_Character) return Boolean;
```

5/3

```
   function Is_Letter (Item : Wide_Character) return Boolean;
```

6/3

```
   function Is_Lower (Item : Wide_Character) return Boolean;
```

7/3

```
   function Is_Upper (Item : Wide_Character) return Boolean;
```

8/3

```
{AI12-0260-1}    function Is_Basic (Item : Wide_Character) return Boolean;
```

8.1/5

```
   function Is_Digit (Item : Wide_Character) return Boolean;
```

9/3

```
   function Is_Decimal_Digit (Item : Wide_Character) return Boolean
      renames Is_Digit;
```

10/3

```
   function Is_Hexadecimal_Digit (Item : Wide_Character) return Boolean;
```

11/3

```
   function Is_Alphanumeric (Item : Wide_Character) return Boolean;
```

12/3

```
   function Is_Special (Item : Wide_Character) return Boolean;
```

13/3

```
   function Is_Line_Terminator (Item : Wide_Character) return Boolean;
```

14/3

```
   function Is_Mark (Item : Wide_Character) return Boolean;
```

15/3

```
   function Is_Other_Format (Item : Wide_Character) return Boolean;
```

16/3

```
   function Is_Punctuation_Connector (Item : Wide_Character) return Boolean;
```

17/3

```
   function Is_Space (Item : Wide_Character) return Boolean;
```

18/3

```
{AI12-0004-1}    function Is_NFKC (Item : Wide_Character) return Boolean;
```

18.1/5

```
   function Is_Graphic (Item : Wide_Character) return Boolean;
```

19/3

```
   function To_Lower (Item : Wide_Character) return Wide_Character;
   function To_Upper (Item : Wide_Character) return Wide_Character;
```

20/3

```
{AI12-0260-1}    function To_Basic (Item : Wide_Character) return Wide_Character;
```

20.1/5

```
   function To_Lower (Item : Wide_String) return Wide_String;
   function To_Upper (Item : Wide_String) return Wide_String;
```

21/3

```
{AI12-0260-1}    function To_Basic (Item : Wide_String) return Wide_String;
```

21.1/5

```
end Ada.Wide_Characters.Handling;
```

22/3

{*AI05-0185-1*} The subprograms defined in Wide_Characters.Handling are locale independent.

23/3

24/3 `function Character_Set_Version return String;`

25/3 {*AI05-0266-1*} Returns an implementation-defined identifier that identifies the version of the character set standard that is used for categorizing characters by the implementation.

26/3 `function Is_Control (Item : Wide_Character) return Boolean;`

27/3 {*AI05-0185-1*} Returns True if the Wide_Character designated by Item is categorized as other_control; otherwise returns False.

28/3 `function Is_Letter (Item : Wide_Character) return Boolean;`

29/3 {*AI05-0185-1*} Returns True if the Wide_Character designated by Item is categorized as letter_uppercase, letter_lowercase, letter_titlecase, letter_modifier, letter_other, or number_letter; otherwise returns False.

30/3 `function Is_Lower (Item : Wide_Character) return Boolean;`

31/3 {*AI05-0185-1*} Returns True if the Wide_Character designated by Item is categorized as letter_lowercase; otherwise returns False.

32/3 `function Is_Upper (Item : Wide_Character) return Boolean;`

33/3 {*AI05-0185-1*} Returns True if the Wide_Character designated by Item is categorized as letter_uppercase; otherwise returns False.

33.1/5 `function Is_Basic (Item : Wide_Character) return Boolean;`

33.2/5 {*AI12-0260-1*} {*AI12-0450-1*} Returns True if the Wide_Character designated by Item has no Decomposition Mapping in the code charts of ISO/IEC 10646:2020; otherwise returns False.

33.a/5 **Implementation Note:** Decomposition Mapping is defined in Clause 33 of ISO/IEC 10646:2020. Machine-readable (and normative!) versions of this can be found as Character Decomposition Mapping, described in file http://www.unicode.org/Public/UCD/latest/ucd/UnicodeData.txt, field 5 (which is the 6th item, Unicode counts from zero).

34/3 `function Is_Digit (Item : Wide_Character) return Boolean;`

35/3 {*AI05-0185-1*} Returns True if the Wide_Character designated by Item is categorized as number_decimal; otherwise returns False.

36/3 `function Is_Hexadecimal_Digit (Item : Wide_Character) return Boolean;`

37/3 {*AI05-0185-1*} Returns True if the Wide_Character designated by Item is categorized as number_decimal, or is in the range 'A' .. 'F' or 'a' .. 'f'; otherwise returns False.

38/3 `function Is_Alphanumeric (Item : Wide_Character) return Boolean;`

39/3 {*AI05-0185-1*} Returns True if the Wide_Character designated by Item is categorized as letter_uppercase, letter_lowercase, letter_titlecase, letter_modifier, letter_other, number_letter, or number_decimal; otherwise returns False.

40/3 `function Is_Special (Item : Wide_Character) return Boolean;`

41/3 {*AI05-0185-1*} Returns True if the Wide_Character designated by Item is categorized as graphic_character, but not categorized as letter_uppercase, letter_lowercase, letter_titlecase, letter_modifier, letter_other, number_letter, or number_decimal; otherwise returns False.

42/3 `function Is_Line_Terminator (Item : Wide_Character) return Boolean;`

43/3 {*AI05-0185-1*} Returns True if the Wide_Character designated by Item is categorized as separator_line or separator_paragraph, or if Item is a conventional line terminator character (Line_Feed, Line_Tabulation, Form_Feed, Carriage_Return, Next_Line); otherwise returns False.

```
function Is_Mark (Item : Wide_Character) return Boolean;
```
44/3

{*AI05-0185-1*} Returns True if the Wide_Character designated by Item is categorized as mark_non_spacing or mark_spacing_combining; otherwise returns False. 45/3

```
function Is_Other_Format (Item : Wide_Character) return Boolean;
```
46/3

{*AI05-0185-1*} Returns True if the Wide_Character designated by Item is categorized as other_format; otherwise returns False. 47/3

```
function Is_Punctuation_Connector (Item : Wide_Character) return Boolean;
```
48/3

{*AI05-0185-1*} Returns True if the Wide_Character designated by Item is categorized as punctuation_connector; otherwise returns False. 49/3

```
function Is_Space (Item : Wide_Character) return Boolean;
```
50/3

{*AI05-0185-1*} Returns True if the Wide_Character designated by Item is categorized as separator_space; otherwise returns False. 51/3

```
function Is_NFKC (Item : Wide_Character) return Boolean;
```
51.1/5

{*AI12-0004-1*} {*AI12-0263-1*} {*AI12-0439-1*} {*AI12-0450-1*} Returns True if the Wide_Character designated by Item can be present in a string normalized to Normalization Form KC (as defined by Clause 22 of ISO/IEC 10646:2020), otherwise returns False. 51.2/5

Reason: Wide_Characters for which this function returns False are not allowed in identifiers (see 2.3) even if they are categorized as letters or digits. 51.a/5

Implementation Note: This function returns False if the Unicode property NFKC Quick Check (NFKC_QC in the files) has the value No. See the Implementation Notes in 2.3 for the source of this property. 51.b/5

Discussion: A string for which Is_NFKC is true for every character may still not be in Normalization Form KC, as Is_NFKC returns true for characters that are dependent on characters around them as to whether they are removed by normalization. Ada does not provide a full normalization operation (it is complex and expensive). 51.c/5

```
function Is_Graphic (Item : Wide_Character) return Boolean;
```
52/3

{*AI05-0185-1*} Returns True if the Wide_Character designated by Item is categorized as graphic_character; otherwise returns False. 53/3

```
function To_Lower (Item : Wide_Character) return Wide_Character;
```
54/3

{*AI05-0185-1*} {*AI05-0266-1*} {*AI05-0299-1*} {*AI12-0263-1*} {*AI12-0450-1*} Returns the Simple Lowercase Mapping as defined by documents referenced in Clause 2 of ISO/IEC 10646:2020 of the Wide_Character designated by Item. If the Simple Lowercase Mapping does not exist for the Wide_Character designated by Item, then the value of Item is returned. 55/5

Discussion: {*AI12-0263-1*} {*AI12-0450-1*} The "documents referenced" means Unicode, Chapter 4 (specifically, section 4.2 — Case). The case mappings come from Unicode as ISO/IEC 10646:2020 does not include complete case mappings. See the Implementation Notes in subclause 1.1.4 for machine-readable versions of both Uppercase and Lowercase mappings. 55.a/5

```
function To_Lower (Item : Wide_String) return Wide_String;
```
56/3

{*AI05-0185-1*} Returns the result of applying the To_Lower conversion to each Wide_Character element of the Wide_String designated by Item. The result is the null Wide_String if the value of the formal parameter is the null Wide_String. The lower bound of the result Wide_String is 1. 57/3

```
function To_Upper (Item : Wide_Character) return Wide_Character;
```
58/3

{*AI05-0185-1*} {*AI05-0266-1*} {*AI05-0299-1*} {*AI12-0263-1*} {*AI12-0450-1*} Returns the Simple Uppercase Mapping as defined by documents referenced in Clause 2 of ISO/IEC 10646:2020 of the Wide_Character designated by Item. If the Simple Uppercase Mapping does not exist for the Wide_Character designated by Item, then the value of Item is returned. 59/5

60/3 `function To_Upper (Item : Wide_String) return Wide_String;`

61/3 {*AI05-0185-1*} Returns the result of applying the To_Upper conversion to each Wide_Character element of the Wide_String designated by Item. The result is the null Wide_String if the value of the formal parameter is the null Wide_String. The lower bound of the result Wide_String is 1.

61.1/5 `function To_Basic (Item : Wide_Character) return Wide_Character;`

61.2/5 {*AI12-0260-1*} {*AI12-0450-1*} Returns the Wide_Character whose code point is given by the first value of its Decomposition Mapping in the code charts of ISO/IEC 10646:2020 if any; returns Item otherwise.

61.3/5 `function To_Basic (Item : Wide_String) return Wide_String;`

61.4/5 {*AI12-0260-1*} Returns the result of applying the To_Basic conversion to each Wide_Character element of the Wide_String designated by Item. The result is the null Wide_String if the value of the formal parameter is the null Wide_String. The lower bound of the result Wide_String is 1.

Implementation Advice

62/3 {*AI05-0266-1*} The string returned by Character_Set_Version should include either “10646:” or “Unicode”.

62.a.1/3 **Implementation Advice:** The string returned by Wide_Characters.Handling.Character_Set_Version should include either “10646:” or “Unicode”.

62.a/5 **Discussion:** {*AI12-0263-1*} {*AI12-0450-1*} The intent is that the returned string include the year for 10646 (as in "10646:2020"), and the version number for Unicode (as in "Unicode 13.0"). We don't try to specify that further so we don't need to decide how to represent Corrigenda for 10646, nor which of these is preferred. (Giving a Unicode version is more accurate, as the case folding and mapping rules always come from a Unicode version [10646 just tells one to look at Unicode to get those], and the character classifications ought to be the same for equivalent versions, but we don't want to talk about non-ISO standards in an ISO standard.)

63/5 NOTE 1 {*AI05-0266-1*} {*AI12-0440-1*} {*AI12-0450-1*} The results returned by these functions can depend on which particular version of ISO/IEC 10646 is supported by the implementation (see 2.1).

64/5 NOTE 2 {*AI05-0286-1*} {*AI12-0449-1*} The case insensitive equality comparison routines provided in A.4.10 are also available for wide strings (see A.4.7).

Extensions to Ada 2005

64.a/3 {*AI05-0185-1*} {*AI05-0266-1*} The package Wide_Characters.Handling is new.

Incompatibilities With Ada 2012

64.b/5 {*AI12-0004-1*} {*AI12-0260-1*} Added additional classification routines Is_Basic and Is_NFKC, and additional conversion routine To_Basic. Therefore, a use clause conflict is possible; see the introduction of Annex A for more on this topic.

A.3.6 The Package Wide_Wide_Characters.Handling

1/3 {*AI05-0185-1*} The package Wide_Wide_Characters.Handling has the same contents as Wide_Characters.Handling except that each occurrence of Wide_Character is replaced by Wide_Wide_Character, and each occurrence of Wide_String is replaced by Wide_Wide_String.

Extensions to Ada 2005

1.a/3 {*AI05-0185-1*} The package Wide_Wide_Characters.Handling is new.

A.4 String Handling

1/3 {*AI95-00285-01*} {*AI05-0299-1*} This subclause presents the specifications of the package Strings and several child packages, which provide facilities for dealing with string data. Fixed-length, bounded-

length, and unbounded-length strings are supported, for String, Wide_String, and Wide_Wide_String. The string-handling subprograms include searches for pattern strings and for characters in program-specified sets, translation (via a character-to-character mapping), and transformation (replacing, inserting, overwriting, and deleting of substrings).

Extensions to Ada 83

{*AI05-0299-1*} This subclause is new to Ada 95. 1.a/3

Wording Changes from Ada 95

{*AI95-00285-01*} Included Wide_Wide_String in this description; the individual changes are documented as extensions as needed. 1.b/2

A.4.1 The Package Strings

The package Strings provides declarations common to the string handling packages. 1

Static Semantics

The library package Strings has the following declaration: 2

```
{AI12-0414-1} package Ada.Strings                                                    3/5
   with Pure is

{AI95-00285-01}    Space      : constant Character      := ' ';                     4/2
   Wide_Space : constant Wide_Character := ' ';
   Wide_Wide_Space : constant Wide_Wide_Character := ' ';

   Length_Error, Pattern_Error, Index_Error, Translation_Error : exception;         5

   type Alignment  is (Left, Right, Center);                                        6
   type Truncation is (Left, Right, Error);
   type Membership is (Inside, Outside);
   type Direction  is (Forward, Backward);
   type Trim_End   is (Left, Right, Both);
end Ada.Strings;
```

Incompatibilities With Ada 95

{*AI95-00285-01*} {*AI05-0005-1*} Constant Wide_Wide_Space is added to Ada.Strings. If Ada.Strings is referenced in a use_clause, and an entity *E* with a defining_identifier of Wide_Wide_Space is defined in a package that is also referenced in a use_clause, the entity *E* may no longer be use-visible, resulting in errors. This should be rare and is easily fixed if it does occur. 6.a/3

A.4.2 The Package Strings.Maps

{*AI12-0445-1*} The package Strings.Maps defines the types, operations, and other entities necessary for character sets and character-to-character mappings. 1/5

Static Semantics

The library package Strings.Maps has the following declaration: 2

```
{AI95-00362-01} {AI12-0414-1} package Ada.Strings.Maps                               3/5
   with Pure is

{AI95-00161-01} {AI12-0399-1}     -- Representation for a set of character values:  4/5
   type Character_Set is private
      with Preelaborable_Initialization;

   Null_Set : constant Character_Set;                                                5
```

```
6        type Character_Range is
           record
              Low  : Character;
              High : Character;
           end record;
         -- Represents Character range Low..High
7        type Character_Ranges is array (Positive range <>) of Character_Range;
8        function To_Set    (Ranges : in Character_Ranges)return Character_Set;
9        function To_Set    (Span   : in Character_Range)return Character_Set;
10       function To_Ranges (Set    : in Character_Set)  return Character_Ranges;
11       function "="   (Left, Right : in Character_Set) return Boolean;
12       function "not" (Right : in Character_Set)       return Character_Set;
         function "and" (Left, Right : in Character_Set) return Character_Set;
         function "or"  (Left, Right : in Character_Set) return Character_Set;
         function "xor" (Left, Right : in Character_Set) return Character_Set;
         function "-"   (Left, Right : in Character_Set) return Character_Set;
13       function Is_In (Element : in Character;
                         Set     : in Character_Set)
            return Boolean;
14       function Is_Subset (Elements : in Character_Set;
                             Set      : in Character_Set)
            return Boolean;
15       function "<=" (Left  : in Character_Set;
                        Right : in Character_Set)
            return Boolean renames Is_Subset;
16       -- Alternative representation for a set of character values:
         subtype Character_Sequence is String;
17       function To_Set (Sequence  : in Character_Sequence)return Character_Set;
18       function To_Set (Singleton : in Character)     return Character_Set;
19       function To_Sequence (Set  : in Character_Set) return Character_Sequence;
20/5  {AI95-00161-01} {AI12-0399-1}    -- Representation for a character to character mapping:
            type Character_Mapping is private
               with Preelaborable_Initialization;
21          function Value (Map     : in Character_Mapping;
                            Element : in Character)
               return Character;
22          Identity : constant Character_Mapping;
23          function To_Mapping (From, To : in Character_Sequence)
               return Character_Mapping;
24          function To_Domain (Map : in Character_Mapping)
               return Character_Sequence;
            function To_Range  (Map : in Character_Mapping)
               return Character_Sequence;
25          type Character_Mapping_Function is
               access function (From : in Character) return Character;
26       private
            ... -- not specified by the language
         end Ada.Strings.Maps;
```

27 An object of type Character_Set represents a set of characters.

28 Null_Set represents the set containing no characters.

29 An object Obj of type Character_Range represents the set of characters in the range Obj.Low .. Obj.High.

30 An object Obj of type Character_Ranges represents the union of the sets corresponding to Obj(I) for I in Obj'Range.

```
function To_Set (Ranges : in Character_Ranges) return Character_Set;
```

{*AI05-0264-1*} If Ranges'Length=0 then Null_Set is returned; otherwise, the returned value represents the set corresponding to Ranges. 32/3

```
function To_Set (Span : in Character_Range) return Character_Set;
```

The returned value represents the set containing each character in Span.

```
function To_Ranges (Set : in Character_Set) return Character_Ranges;
```

{*AI05-0264-1*} If Set = Null_Set, then an empty Character_Ranges array is returned; otherwise, the shortest array of contiguous ranges of Character values in Set, in increasing order of Low, is returned. 36/3

```
function "=" (Left, Right : in Character_Set) return Boolean;
```

The function "=" returns True if Left and Right represent identical sets, and False otherwise.

Each of the logical operators "**not**", "**and**", "**or**", and "**xor**" returns a Character_Set value that represents the set obtained by applying the corresponding operation to the set(s) represented by the parameter(s) of the operator. "–"(Left, Right) is equivalent to "and"(Left, "not"(Right)).

Reason: The set minus operator is provided for efficiency.

```
function Is_In (Element : in Character;
                Set     : in Character_Set);
   return Boolean;
```

Is_In returns True if Element is in Set, and False otherwise.

```
function Is_Subset (Elements : in Character_Set;
                    Set      : in Character_Set)
   return Boolean;
```

Is_Subset returns True if Elements is a subset of Set, and False otherwise.

```
subtype Character_Sequence is String;
```

The Character_Sequence subtype is used to portray a set of character values and also to identify the domain and range of a character mapping.

Reason: Although a named subtype is redundant — the predefined type String could have been used for the parameter to To_Set and To_Mapping below — the use of a differently named subtype identifies the intended purpose of the parameter.

```
function To_Set (Sequence  : in Character_Sequence) return Character_Set;

function To_Set (Singleton : in Character)          return Character_Set;
```

Sequence portrays the set of character values that it explicitly contains (ignoring duplicates). Singleton portrays the set comprising a single Character. Each of the To_Set functions returns a Character_Set value that represents the set portrayed by Sequence or Singleton.

```
function To_Sequence (Set : in Character_Set) return Character_Sequence;
```

The function To_Sequence returns a Character_Sequence value containing each of the characters in the set represented by Set, in ascending order with no duplicates.

```
type Character_Mapping is private;
```

An object of type Character_Mapping represents a Character-to-Character mapping.

52
```
function Value (Map     : in Character_Mapping;
                Element : in Character)
   return Character;
```

53 The function Value returns the Character value to which Element maps with respect to the mapping represented by Map.

54 A character C *matches* a pattern character P with respect to a given Character_Mapping value Map if Value(Map, C) = P. A string S *matches* a pattern string P with respect to a given Character_Mapping if their lengths are the same and if each character in S matches its corresponding character in the pattern string P.

54.a **Discussion:** In an earlier version of the string handling packages, the definition of matching was symmetrical, namely C matches P if Value(Map,C) = Value(Map,P). However, applying the mapping to the pattern was confusing according to some reviewers. Furthermore, if the symmetrical version is needed, it can be achieved by applying the mapping to the pattern (via translation) prior to passing it as a parameter.

55 String handling subprograms that deal with character mappings have parameters whose type is Character_Mapping.

56
```
Identity : constant Character_Mapping;
```

57 Identity maps each Character to itself.

58
```
function To_Mapping (From, To : in Character_Sequence)
    return Character_Mapping;
```

59 To_Mapping produces a Character_Mapping such that each element of From maps to the corresponding element of To, and each other character maps to itself. If From'Length /= To'Length, or if some character is repeated in From, then Translation_Error is propagated.

60
```
function To_Domain (Map : in Character_Mapping) return Character_Sequence;
```

61 To_Domain returns the shortest Character_Sequence value D such that each character not in D maps to itself, and such that the characters in D are in ascending order. The lower bound of D is 1.

62
```
function To_Range  (Map : in Character_Mapping) return Character_Sequence;
```

63/1 {*8652/0048*} {*AI95-00151-01*} To_Range returns the Character_Sequence value R, such that if D = To_Domain(Map), then R has the same bounds as D, and D(I) maps to R(I) for each I in D'Range.

64 An object F of type Character_Mapping_Function maps a Character value C to the Character value F.**all**(C), which is said to *match* C with respect to mapping function F.

65 NOTE 1 Character_Mapping and Character_Mapping_Function are used both for character equivalence mappings in the search subprograms (such as for case insensitivity) and as transformational mappings in the Translate subprograms.

66 NOTE 2 To_Domain(Identity) and To_Range(Identity) each returns the null string.

66.a **Reason:** Package Strings.Maps is not pure, since it declares an access-to-subprogram type.

Examples

67/5 {*AI12-0429-1*} *Example of use of Strings.Maps.To_Mapping:*

68 To_Mapping("ABCD", "ZZAB") returns a Character_Mapping that maps 'A' and 'B' to 'Z', 'C' to 'A', 'D' to 'B', and each other Character to itself.

Extensions to Ada 95

68.a/2 {*AI95-00161-01*} **Amendment Correction:** Added pragma Preelaborable_Initialization to types Character_Set and Character_Mapping, so that they can be used to declare default-initialized objects in preelaborated units.

{*AI95-00362-01*} Strings.Maps is now Pure, so it can be used in pure units. 68.b/2

Wording Changes from Ada 95

{*8652/0048*} {*AI95-00151-01*} **Corrigendum:** Corrected the definition of the range of the result of To_Range, since the Ada 95 definition makes no sense. 68.c/2

A.4.3 Fixed-Length String Handling

The language-defined package Strings.Fixed provides string-handling subprograms for fixed-length strings; that is, for values of type Standard.String. Several of these subprograms are procedures that modify the contents of a String that is passed as an **out** or an **in out** parameter; each has additional parameters to control the effect when the logical length of the result differs from the parameter's length. 1

For each function that returns a String, the lower bound of the returned value is 1. 2

Discussion: {*AI95-00114-01*} Most operations that yield a String are provided both as a function and as a procedure. The functional form is possibly a more aesthetic style but may introduce overhead due to extra copying or dynamic memory usage in some implementations. Thus a procedural form, with an **in out** parameter so that all copying is done `in place', is also supplied. 2.a/2

The basic model embodied in the package is that a fixed-length string comprises significant characters and possibly padding (with space characters) on either or both ends. When a shorter string is copied to a longer string, padding is inserted, and when a longer string is copied to a shorter one, padding is stripped. The Move procedure in Strings.Fixed, which takes a String as an **out** parameter, allows the programmer to control these effects. Similar control is provided by the string transformation procedures. 3

Static Semantics

The library package Strings.Fixed has the following declaration: 4

```
{AI12-0241-1} {AI12-0302-1} with Ada.Strings.Maps;                                  5/5
package Ada.Strings.Fixed
   with Preelaborate, Nonblocking, Global => in out synchronized is
-- "Copy" procedure for strings of possibly different lengths                       6
   procedure Move (Source  : in  String;                                            7
                   Target  : out String;
                   Drop    : in  Truncation := Error;
                   Justify : in  Alignment  := Left;
                   Pad     : in  Character  := Space);
-- Search subprograms                                                               8
{AI95-00301-01}    function Index (Source  : in String;                             8.1/2
                   Pattern : in String;
                   From    : in Positive;
                   Going   : in Direction := Forward;
                   Mapping : in Maps.Character_Mapping := Maps.Identity)
      return Natural;
{AI95-00301-01}    function Index (Source  : in String;                             8.2/2
                   Pattern : in String;
                   From    : in Positive;
                   Going   : in Direction := Forward;
                   Mapping : in Maps.Character_Mapping_Function)
      return Natural;
   function Index (Source   : in String;                                            9
                   Pattern  : in String;
                   Going    : in Direction := Forward;
                   Mapping  : in Maps.Character_Mapping
                              := Maps.Identity)
      return Natural;
```

```
10          function Index (Source   : in String;
                            Pattern  : in String;
                            Going    : in Direction := Forward;
                            Mapping  : in Maps.Character_Mapping_Function)
               return Natural;

10.1/2   {AI95-00301-01}    function Index (Source  : in String;
                            Set     : in Maps.Character_Set;
                            From    : in Positive;
                            Test    : in Membership := Inside;
                            Going   : in Direction := Forward)
               return Natural;

11          function Index (Source : in String;
                            Set    : in Maps.Character_Set;
                            Test   : in Membership := Inside;
                            Going  : in Direction  := Forward)
               return Natural;

11.1/2   {AI95-00301-01}    function Index_Non_Blank (Source : in String;
                                      From   : in Positive;
                                      Going  : in Direction := Forward)
               return Natural;

12          function Index_Non_Blank (Source : in String;
                                      Going  : in Direction := Forward)
               return Natural;

13          function Count (Source   : in String;
                            Pattern  : in String;
                            Mapping  : in Maps.Character_Mapping
                                         := Maps.Identity)
               return Natural;

14          function Count (Source   : in String;
                            Pattern  : in String;
                            Mapping  : in Maps.Character_Mapping_Function)
               return Natural;

15          function Count (Source   : in String;
                            Set      : in Maps.Character_Set)
               return Natural;

15.1/3   {AI05-0031-1}    procedure Find_Token (Source : in String;
                                  Set    : in Maps.Character_Set;
                                  From   : in Positive;
                                  Test   : in Membership;
                                  First  : out Positive;
                                  Last   : out Natural);

16          procedure Find_Token (Source : in String;
                                  Set    : in Maps.Character_Set;
                                  Test   : in Membership;
                                  First  : out Positive;
                                  Last   : out Natural);

17       -- String translation subprograms

18          function Translate (Source  : in String;
                                Mapping : in Maps.Character_Mapping)
               return String;

19          procedure Translate (Source  : in out String;
                                 Mapping : in Maps.Character_Mapping);

20          function Translate (Source  : in String;
                                Mapping : in Maps.Character_Mapping_Function)
               return String;

21          procedure Translate (Source  : in out String;
                                 Mapping : in Maps.Character_Mapping_Function);
```

```
-- String transformation subprograms                                    22
   function Replace_Slice (Source   : in String;                        23
                           Low      : in Positive;
                           High     : in Natural;
                           By       : in String)
      return String;
   procedure Replace_Slice (Source   : in out String;                   24
                            Low      : in Positive;
                            High     : in Natural;
                            By       : in String;
                            Drop     : in Truncation := Error;
                            Justify  : in Alignment  := Left;
                            Pad      : in Character  := Space);
   function Insert (Source   : in String;                               25
                    Before   : in Positive;
                    New_Item : in String)
      return String;
   procedure Insert (Source   : in out String;                          26
                     Before   : in Positive;
                     New_Item : in String;
                     Drop     : in Truncation := Error);
   function Overwrite (Source   : in String;                            27
                       Position : in Positive;
                       New_Item : in String)
      return String;
   procedure Overwrite (Source   : in out String;                       28
                        Position : in Positive;
                        New_Item : in String;
                        Drop     : in Truncation := Right);
   function Delete (Source  : in String;                                29
                    From    : in Positive;
                    Through : in Natural)
      return String;
   procedure Delete (Source  : in out String;                           30
                     From    : in Positive;
                     Through : in Natural;
                     Justify : in Alignment := Left;
                     Pad     : in Character := Space);
--String selector subprograms                                           31
   function Trim (Source : in String;
                  Side   : in Trim_End)
      return String;
   procedure Trim (Source  : in out String;                             32
                   Side    : in Trim_End;
                   Justify : in Alignment := Left;
                   Pad     : in Character := Space);
   function Trim (Source : in String;                                   33
                  Left   : in Maps.Character_Set;
                  Right  : in Maps.Character_Set)
      return String;
   procedure Trim (Source  : in out String;                             34
                   Left    : in Maps.Character_Set;
                   Right   : in Maps.Character_Set;
                   Justify : in Alignment := Strings.Left;
                   Pad     : in Character := Space);
   function Head (Source : in String;                                   35
                  Count  : in Natural;
                  Pad    : in Character := Space)
      return String;
```

```
36        procedure Head (Source  : in out String;
                          Count   : in Natural;
                          Justify : in Alignment := Left;
                          Pad     : in Character := Space);
37        function Tail (Source : in String;
                         Count  : in Natural;
                         Pad    : in Character := Space)
             return String;
38        procedure Tail (Source  : in out String;
                          Count   : in Natural;
                          Justify : in Alignment := Left;
                          Pad     : in Character := Space);
39     --String constructor functions
40        function "*" (Left  : in Natural;
                        Right : in Character) return String;
41        function "*" (Left  : in Natural;
                        Right : in String) return String;
42     end Ada.Strings.Fixed;
```

43 The effects of the above subprograms are as follows.

```
44     procedure Move (Source  : in  String;
                       Target  : out String;
                       Drop    : in  Truncation := Error;
                       Justify : in  Alignment  := Left;
                       Pad     : in  Character  := Space);
```

45/3 {*AI05-0264-1*} The Move procedure copies characters from Source to Target. If Source has the same length as Target, then the effect is to assign Source to Target. If Source is shorter than Target, then:

46 • If Justify=Left, then Source is copied into the first Source'Length characters of Target.

47 • If Justify=Right, then Source is copied into the last Source'Length characters of Target.

48 • If Justify=Center, then Source is copied into the middle Source'Length characters of Target. In this case, if the difference in length between Target and Source is odd, then the extra Pad character is on the right.

49 • Pad is copied to each Target character not otherwise assigned.

50 If Source is longer than Target, then the effect is based on Drop.

51 • If Drop=Left, then the rightmost Target'Length characters of Source are copied into Target.

52 • If Drop=Right, then the leftmost Target'Length characters of Source are copied into Target.

53 • If Drop=Error, then the effect depends on the value of the Justify parameter and also on whether any characters in Source other than Pad would fail to be copied:

54 • If Justify=Left, and if each of the rightmost Source'Length-Target'Length characters in Source is Pad, then the leftmost Target'Length characters of Source are copied to Target.

55 • If Justify=Right, and if each of the leftmost Source'Length-Target'Length characters in Source is Pad, then the rightmost Target'Length characters of Source are copied to Target.

56 • Otherwise, Length_Error is propagated.

56.a **Ramification:** The Move procedure will work even if Source and Target overlap.

Reason: The order of parameters (Source before Target) corresponds to the order in COBOL's MOVE verb. 56.b

```
function Index (Source  : in String;                                    56.1/2
                Pattern : in String;
                From    : in Positive;
                Going   : in Direction := Forward;
                Mapping : in Maps.Character_Mapping := Maps.Identity)
   return Natural;

function Index (Source  : in String;
                Pattern : in String;
                From    : in Positive;
                Going   : in Direction := Forward;
                Mapping : in Maps.Character_Mapping_Function)
   return Natural;
```

{*AI95-00301-01*} {*AI05-0056-1*} Each Index function searches, starting from From, for a slice of Source, with length Pattern'Length, that matches Pattern with respect to Mapping; the parameter Going indicates the direction of the lookup. If Source is the null string, Index returns 0; otherwise, if From is not in Source'Range, then Index_Error is propagated. If Going = Forward, then Index returns the smallest index I which is greater than or equal to From such that the slice of Source starting at I matches Pattern. If Going = Backward, then Index returns the largest index I such that the slice of Source starting at I matches Pattern and has an upper bound less than or equal to From. If there is no such slice, then 0 is returned. If Pattern is the null string, then Pattern_Error is propagated. 56.2/3

Discussion: There is no default parameter for From; the default value would need to depend on other parameters (the bounds of Source and the direction Going). It is better to use overloaded functions rather than a special value to represent the default. 56.c/2

There is no default value for the Mapping parameter that is a Character_Mapping_Function; if there were, a call would be ambiguous since there is also a default for the Mapping parameter that is a Character_Mapping. 56.d/2

{*AI05-0056-1*} The language does not define when the Pattern_Error check is made. (That's because many common searching implementations require a nonempty pattern) That means that the result for a call like `Index ("", "")` could be 0 or could raise Pattern_Error. Similarly, in the call `Index ("", "", From => 2)`, the language does not define whether Pattern_Error or Index_Error is raised. 56.e/3

```
function Index (Source   : in String;                                   57
                Pattern  : in String;
                Going    : in Direction := Forward;
                Mapping  : in Maps.Character_Mapping
                              := Maps.Identity)
   return Natural;

function Index (Source   : in String;
                Pattern  : in String;
                Going    : in Direction := Forward;
                Mapping  : in Maps.Character_Mapping_Function)
   return Natural;
```

{*AI95-00301-01*} If Going = Forward, returns 58/2

```
   Index (Source, Pattern, Source'First, Forward, Mapping);             58.1/2
```

{*AI05-0264-1*} otherwise, returns 58.2/3

```
   Index (Source, Pattern, Source'Last, Backward, Mapping);             58.3/2
```

*This paragraph was deleted.*There is no default value for the Mapping parameter that is a Character_Mapping_Function; if there were, a call would be ambiguous since there is also a default for the Mapping parameter that is a Character_Mapping. 58.a/2

58.4/2
```
function Index (Source  : in String;
                Set     : in Maps.Character_Set;
                From    : in Positive;
                Test    : in Membership := Inside;
                Going   : in Direction := Forward)
   return Natural;
```

58.5/3 {*AI95-00301-01*} {*AI05-0056-1*} Index searches for the first or last occurrence of any of a set of characters (when Test=Inside), or any of the complement of a set of characters (when Test=Outside). If Source is the null string, Index returns 0; otherwise, if From is not in Source'Range, then Index_Error is propagated. Otherwise, it returns the smallest index I >= From (if Going=Forward) or the largest index I <= From (if Going=Backward) such that Source(I) satisfies the Test condition with respect to Set; it returns 0 if there is no such Character in Source.

59
```
function Index (Source : in String;
                Set    : in Maps.Character_Set;
                Test   : in Membership := Inside;
                Going  : in Direction  := Forward)
   return Natural;
```

60/2 {*AI95-00301-01*} If Going = Forward, returns

60.1/2
```
   Index (Source, Set, Source'First, Test, Forward);
```

60.2/3 {*AI05-0264-1*} otherwise, returns

60.3/2
```
   Index (Source, Set, Source'Last, Test, Backward);
```

60.4/2
```
function Index_Non_Blank (Source : in String;
                          From   : in Positive;
                          Going  : in Direction := Forward)
   return Natural;
```

60.5/2 {*AI95-00301-01*} Returns Index (Source, Maps.To_Set(Space), From, Outside, Going);

61
```
function Index_Non_Blank (Source : in String;
                          Going  : in Direction := Forward)
   return Natural;
```

62 Returns Index(Source, Maps.To_Set(Space), Outside, Going)

63
```
function Count (Source   : in String;
                Pattern  : in String;
                Mapping  : in Maps.Character_Mapping
                             := Maps.Identity)
   return Natural;

function Count (Source   : in String;
                Pattern  : in String;
                Mapping  : in Maps.Character_Mapping_Function)
   return Natural;
```

64 Returns the maximum number of nonoverlapping slices of Source that match Pattern with respect to Mapping. If Pattern is the null string then Pattern_Error is propagated.

64.a **Reason:** We say `maximum number' because it is possible to slice a source string in different ways yielding different numbers of matches. For example if Source is "ABABABA" and Pattern is "ABA", then Count yields 2, although there is a partitioning of Source that yields just 1 match, for the middle slice. Saying `maximum number' is equivalent to saying that the pattern match starts either at the low index or the high index position.

65
```
function Count (Source   : in String;
                Set      : in Maps.Character_Set)
   return Natural;
```

66 Returns the number of occurrences in Source of characters that are in Set.

```
procedure Find_Token (Source : in String;                                    66.1/3
                      Set    : in Maps.Character_Set;
                      From   : in Positive;
                      Test   : in Membership;
                      First  : out Positive;
                      Last   : out Natural);
```

{*AI05-0031-1*} If Source is not the null string and From is not in Source'Range, then Index_Error is raised. Otherwise, First is set to the index of the first character in Source(From .. Source'Last) that satisfies the Test condition. Last is set to the largest index such that all characters in Source(First .. Last) satisfy the Test condition. If no characters in Source(From .. Source'Last) satisfy the Test condition, First is set to From, and Last is set to 0. 66.2/3

```
procedure Find_Token (Source : in String;                                    67
                      Set    : in Maps.Character_Set;
                      Test   : in Membership;
                      First  : out Positive;
                      Last   : out Natural);
```

{*8652/0049*} {*AI95-00128-01*} {*AI05-0031-1*} Equivalent to Find_Token (Source, Set, Source'First, Test, First, Last). 68/3

Ramification: {*AI05-0031-1*} If Source'First is not in Positive, which can only happen for an empty string, this will raise Constraint_Error. 68.a/3

```
function Translate (Source  : in String;                                     69
                    Mapping : in Maps.Character_Mapping)
   return String;

function Translate (Source  : in String;
                    Mapping : in Maps.Character_Mapping_Function)
   return String;
```

Returns the string S whose length is Source'Length and such that S(I) is the character to which Mapping maps the corresponding element of Source, for I in 1..Source'Length. 70

```
procedure Translate (Source  : in out String;                                71
                     Mapping : in Maps.Character_Mapping);

procedure Translate (Source  : in out String;
                     Mapping : in Maps.Character_Mapping_Function);
```

Equivalent to Source := Translate(Source, Mapping). 72

```
function Replace_Slice (Source   : in String;                                73
                        Low      : in Positive;
                        High     : in Natural;
                        By       : in String)
   return String;
```

{*8652/0049*} {*AI95-00128-01*} If Low > Source'Last+1, or High < Source'First–1, then Index_Error is propagated. Otherwise: 74/1

- {*8652/0049*} {*AI95-00128-01*} If High >= Low, then the returned string comprises Source(Source'First..Low–1) & By & Source(High+1..Source'Last), but with lower bound 1. 74.1/1
- {*8652/0049*} {*AI95-00128-01*} If High < Low, then the returned string is Insert(Source, Before=>Low, New_Item=>By). 74.2/1

75
```
procedure Replace_Slice (Source   : in out String;
                         Low      : in Positive;
                         High     : in Natural;
                         By       : in String;
                         Drop     : in Truncation := Error;
                         Justify  : in Alignment  := Left;
                         Pad      : in Character  := Space);
```

76 Equivalent to Move(Replace_Slice(Source, Low, High, By), Source, Drop, Justify, Pad).

77
```
function Insert (Source   : in String;
                 Before   : in Positive;
                 New_Item : in String)
   return String;
```

78/3 {*AI05-0264-1*} Propagates Index_Error if Before is not in Source'First .. Source'Last+1; otherwise, returns Source(Source'First..Before–1) & New_Item & Source(Before..Source'Last), but with lower bound 1.

79
```
procedure Insert (Source   : in out String;
                  Before   : in Positive;
                  New_Item : in String;
                  Drop     : in Truncation := Error);
```

80 Equivalent to Move(Insert(Source, Before, New_Item), Source, Drop).

81
```
function Overwrite (Source   : in String;
                    Position : in Positive;
                    New_Item : in String)
   return String;
```

82/3 {*AI05-0264-1*} Propagates Index_Error if Position is not in Source'First .. Source'Last+1; otherwise, returns the string obtained from Source by consecutively replacing characters starting at Position with corresponding characters from New_Item. If the end of Source is reached before the characters in New_Item are exhausted, the remaining characters from New_Item are appended to the string.

83
```
procedure Overwrite (Source   : in out String;
                     Position : in Positive;
                     New_Item : in String;
                     Drop     : in Truncation := Right);
```

84 Equivalent to Move(Overwrite(Source, Position, New_Item), Source, Drop).

85
```
function Delete (Source  : in String;
                 From    : in Positive;
                 Through : in Natural)
   return String;
```

86/3 {*8652/0049*} {*AI95-00128-01*} {*AI05-0264-1*} If From <= Through, the returned string is Replace_Slice(Source, From, Through, ""); otherwise, it is Source with lower bound 1.

87
```
procedure Delete (Source  : in out String;
                  From    : in Positive;
                  Through : in Natural;
                  Justify : in Alignment := Left;
                  Pad     : in Character := Space);
```

88 Equivalent to Move(Delete(Source, From, Through), Source, Justify => Justify, Pad => Pad).

```
function Trim (Source : in String;                                    89
               Side   : in Trim_End)
  return String;
```

Returns the string obtained by removing from Source all leading Space characters (if Side = Left), all trailing Space characters (if Side = Right), or all leading and trailing Space characters (if Side = Both). 90

```
procedure Trim (Source  : in out String;                              91
                Side    : in Trim_End;
                Justify : in Alignment := Left;
                Pad     : in Character := Space);
```

Equivalent to Move(Trim(Source, Side), Source, Justify=>Justify, Pad=>Pad). 92

```
function Trim (Source : in String;                                    93
               Left   : in Maps.Character_Set;
               Right  : in Maps.Character_Set)
   return String;
```

Returns the string obtained by removing from Source all leading characters in Left and all trailing characters in Right. 94

```
procedure Trim (Source  : in out String;                              95
                Left    : in Maps.Character_Set;
                Right   : in Maps.Character_Set;
                Justify : in Alignment := Strings.Left;
                Pad     : in Character := Space);
```

Equivalent to Move(Trim(Source, Left, Right), Source, Justify => Justify, Pad=>Pad). 96

```
function Head (Source : in String;                                    97
               Count  : in Natural;
               Pad    : in Character := Space)
   return String;
```

{*AI05-0264-1*} Returns a string of length Count. If Count <= Source'Length, the string comprises the first Count characters of Source. Otherwise, its contents are Source concatenated with Count–Source'Length Pad characters. 98/3

```
procedure Head (Source  : in out String;                              99
                Count   : in Natural;
                Justify : in Alignment := Left;
                Pad     : in Character := Space);
```

Equivalent to Move(Head(Source, Count, Pad), Source, Drop=>Error, Justify=>Justify, Pad=>Pad). 100

```
function Tail (Source : in String;                                    101
               Count  : in Natural;
               Pad    : in Character := Space)
   return String;
```

{*AI05-0264-1*} Returns a string of length Count. If Count <= Source'Length, the string comprises the last Count characters of Source. Otherwise, its contents are Count-Source'Length Pad characters concatenated with Source. 102/3

```
procedure Tail (Source  : in out String;                              103
                Count   : in Natural;
                Justify : in Alignment := Left;
                Pad     : in Character := Space);
```

Equivalent to Move(Tail(Source, Count, Pad), Source, Drop=>Error, Justify=>Justify, Pad=>Pad). 104

105
```
function "*" (Left  : in Natural;
              Right : in Character) return String;

function "*" (Left  : in Natural;
              Right : in String) return String;
```

106/1 {*8652/0049*} {*AI95-00128-01*} These functions replicate a character or string a specified number of times. The first function returns a string whose length is Left and each of whose elements is Right. The second function returns a string whose length is Left*Right'Length and whose value is the null string if Left = 0 and otherwise is (Left–1)*Right & Right with lower bound 1.

107/5 NOTE 1 {*AI05-0264-1*} {*AI12-0442-1*} In the Index and Count functions taking Pattern and Mapping parameters, for there to be a match, the actual String parameter passed to Pattern can contain only characters occurring as target characters of the mapping.

108 NOTE 2 In the Insert subprograms, inserting at the end of a string is obtained by passing Source'Last+1 as the Before parameter.

109 NOTE 3 If a null Character_Mapping_Function is passed to any of the string handling subprograms, Constraint_Error is propagated.

Incompatibilities With Ada 95

109.a/3 {*AI95-00301-01*} {*AI05-0005-1*} Overloaded versions of Index and Index_Non_Blank are added to Strings.Fixed. If Strings.Fixed is referenced in a use_clause, and an entity *E* with a defining_identifier of Index or Index_Non_Blank is defined in a package that is also referenced in a use_clause, the entity *E* may no longer be use-visible, resulting in errors. This should be rare and is easily fixed if it does occur.

Wording Changes from Ada 95

109.b/2 {*8652/0049*} {*AI95-00128-01*} **Corrigendum:** Clarified that Find_Token may raise Constraint_Error if Source'First is not in Positive (which is only possible for a null string).

109.c/2 {*8652/0049*} {*AI95-00128-01*} **Corrigendum:** Clarified that Replace_Slice, Delete, and "*" always return a string with lower bound 1.

Incompatibilities With Ada 2005

109.d/3 {*AI05-0031-1*} An overloaded version of Find_Token is added to Strings.Fixed. If Strings.Fixed is referenced in a use_clause, and an entity *E* with a defining_identifier of Find_Token is defined in a package that is also referenced in a use_clause, the entity *E* may no longer be use-visible, resulting in errors. This should be rare and is easily fixed if it does occur.

Wording Changes from Ada 2005

109.e/3 {*AI05-0056-1*} **Correction:** Clarified that Index never raises Index_Error if the source string is null.

A.4.4 Bounded-Length String Handling

1/5 {*AI12-0445-1*} The language-defined package Strings.Bounded provides a generic package each of whose instances yields a private type Bounded_String and a set of operations. An object of a particular Bounded_String type represents a String whose low bound is 1 and whose length can vary conceptually between 0 and a maximum size established at the generic instantiation. The subprograms for fixed-length string handling are either overloaded directly for Bounded_String, or are modified as necessary to reflect the variability in length. Additionally, since the Bounded_String type is private, appropriate constructor and selector operations are provided.

1.a **Reason:** Strings.Bounded declares an inner generic package, versus itself being directly a generic child of Strings, in order to retain compatibility with a version of the string-handling packages that is generic with respect to the character and string types.

1.b **Reason:** The bound of a bounded-length string is specified as a parameter to a generic, versus as the value for a discriminant, because of the inappropriateness of assignment and equality of discriminated types for the copying and comparison of bounded strings.

Static Semantics

The library package Strings.Bounded has the following declaration: 2

```
{AI12-0241-1} {AI12-0302-1} with Ada.Strings.Maps;                                    3/5
package Ada.Strings.Bounded
   with Preelaborate, Nonblocking, Global => in out synchronized is

   generic                                                                            4
      Max   : Positive;     -- Maximum length of a Bounded_String
   package Generic_Bounded_Length is

      Max_Length : constant Positive := Max;                                          5

      type Bounded_String is private;                                                 6

      Null_Bounded_String : constant Bounded_String;                                  7

      subtype Length_Range is Natural range 0 .. Max_Length;                          8

      function Length (Source : in Bounded_String) return Length_Range;               9

   -- Conversion, Concatenation, and Selection functions                             10

      function To_Bounded_String (Source : in String;                                 11
                                  Drop   : in Truncation := Error)
         return Bounded_String;

      function To_String (Source : in Bounded_String) return String;                 12

{AI95-00301-01}       procedure Set_Bounded_String                                    12.1/2
         (Target :    out Bounded_String;
          Source : in     String;
          Drop   : in     Truncation := Error);

      function Append (Left, Right : in Bounded_String;                               13
                       Drop        : in Truncation  := Error)
         return Bounded_String;

      function Append (Left  : in Bounded_String;                                     14
                       Right : in String;
                       Drop  : in Truncation := Error)
         return Bounded_String;

      function Append (Left  : in String;                                             15
                       Right : in Bounded_String;
                       Drop  : in Truncation := Error)
         return Bounded_String;

      function Append (Left  : in Bounded_String;                                     16
                       Right : in Character;
                       Drop  : in Truncation := Error)
         return Bounded_String;

      function Append (Left  : in Character;                                          17
                       Right : in Bounded_String;
                       Drop  : in Truncation := Error)
         return Bounded_String;

      procedure Append (Source   : in out Bounded_String;                             18
                        New_Item : in Bounded_String;
                        Drop     : in Truncation  := Error);

      procedure Append (Source   : in out Bounded_String;                             19
                        New_Item : in String;
                        Drop     : in Truncation  := Error);

      procedure Append (Source   : in out Bounded_String;                             20
                        New_Item : in Character;
                        Drop     : in Truncation  := Error);

      function "&" (Left, Right : in Bounded_String)                                  21
         return Bounded_String;

      function "&" (Left : in Bounded_String; Right : in String)                      22
         return Bounded_String;
```

```
23          function "&" (Left : in String; Right : in Bounded_String)
               return Bounded_String;
24          function "&" (Left : in Bounded_String; Right : in Character)
               return Bounded_String;
25          function "&" (Left : in Character; Right : in Bounded_String)
               return Bounded_String;
26          function Element (Source : in Bounded_String;
                              Index  : in Positive)
               return Character;
27          procedure Replace_Element (Source : in out Bounded_String;
                                       Index  : in Positive;
                                       By     : in Character);
28          function Slice (Source : in Bounded_String;
                            Low    : in Positive;
                            High   : in Natural)
               return String;
28.1/2   {AI95-00301-01}       function Bounded_Slice
            (Source : in Bounded_String;
             Low    : in Positive;
             High   : in Natural)
                return Bounded_String;
28.2/2   {AI95-00301-01}       procedure Bounded_Slice
            (Source : in     Bounded_String;
             Target :    out Bounded_String;
             Low    : in     Positive;
             High   : in     Natural);
29          function "="  (Left, Right : in Bounded_String) return Boolean;
            function "="  (Left : in Bounded_String; Right : in String)
              return Boolean;
30          function "="  (Left : in String; Right : in Bounded_String)
              return Boolean;
31          function "<"  (Left, Right : in Bounded_String) return Boolean;
32          function "<"  (Left : in Bounded_String; Right : in String)
              return Boolean;
33          function "<"  (Left : in String; Right : in Bounded_String)
              return Boolean;
34          function "<=" (Left, Right : in Bounded_String) return Boolean;
35          function "<="  (Left : in Bounded_String; Right : in String)
              return Boolean;
36          function "<="  (Left : in String; Right : in Bounded_String)
              return Boolean;
37          function ">"  (Left, Right : in Bounded_String) return Boolean;
38          function ">"  (Left : in Bounded_String; Right : in String)
              return Boolean;
39          function ">"  (Left : in String; Right : in Bounded_String)
              return Boolean;
40          function ">=" (Left, Right : in Bounded_String) return Boolean;
41          function ">="  (Left : in Bounded_String; Right : in String)
              return Boolean;
42          function ">="  (Left : in String; Right : in Bounded_String)
              return Boolean;
43/2     {AI95-00301-01}     -- Search subprograms
```

```
{AI95-00301-01}       function Index (Source  : in Bounded_String;                    43.1/2
                      Pattern : in String;
                      From    : in Positive;
                      Going   : in Direction := Forward;
                      Mapping : in Maps.Character_Mapping := Maps.Identity)
         return Natural;
{AI95-00301-01}       function Index (Source  : in Bounded_String;                    43.2/2
                      Pattern : in String;
                      From    : in Positive;
                      Going   : in Direction := Forward;
                      Mapping : in Maps.Character_Mapping_Function)
         return Natural;
      function Index (Source   : in Bounded_String;                                   44
                      Pattern  : in String;
                      Going    : in Direction := Forward;
                      Mapping  : in Maps.Character_Mapping
                                 := Maps.Identity)
         return Natural;
      function Index (Source   : in Bounded_String;                                   45
                      Pattern  : in String;
                      Going    : in Direction := Forward;
                      Mapping  : in Maps.Character_Mapping_Function)
         return Natural;
{AI95-00301-01}       function Index (Source  : in Bounded_String;                    45.1/2
                      Set     : in Maps.Character_Set;
                      From    : in Positive;
                      Test    : in Membership := Inside;
                      Going   : in Direction := Forward)
         return Natural;
      function Index (Source : in Bounded_String;                                     46
                      Set    : in Maps.Character_Set;
                      Test   : in Membership := Inside;
                      Going  : in Direction  := Forward)
         return Natural;
{AI95-00301-01}       function Index_Non_Blank (Source : in Bounded_String;           46.1/2
                                From   : in Positive;
                                Going  : in Direction := Forward)
         return Natural;
      function Index_Non_Blank (Source : in Bounded_String;                           47
                                Going  : in Direction := Forward)
         return Natural;
      function Count (Source   : in Bounded_String;                                   48
                      Pattern  : in String;
                      Mapping  : in Maps.Character_Mapping
                                    := Maps.Identity)
         return Natural;
      function Count (Source   : in Bounded_String;                                   49
                      Pattern  : in String;
                      Mapping  : in Maps.Character_Mapping_Function)
         return Natural;
      function Count (Source   : in Bounded_String;                                   50
                      Set      : in Maps.Character_Set)
         return Natural;
{AI05-0031-1}       procedure Find_Token (Source : in Bounded_String;                 50.1/3
                             Set    : in Maps.Character_Set;
                             From   : in Positive;
                             Test   : in Membership;
                             First  : out Positive;
                             Last   : out Natural);
```

```
51          procedure Find_Token (Source : in Bounded_String;
                                  Set    : in Maps.Character_Set;
                                  Test   : in Membership;
                                  First  : out Positive;
                                  Last   : out Natural);
52       -- String translation subprograms
53          function Translate (Source  : in Bounded_String;
                                Mapping : in Maps.Character_Mapping)
             return Bounded_String;
54          procedure Translate (Source  : in out Bounded_String;
                                 Mapping : in Maps.Character_Mapping);
55          function Translate (Source  : in Bounded_String;
                                Mapping : in Maps.Character_Mapping_Function)
             return Bounded_String;
56          procedure Translate (Source  : in out Bounded_String;
                                 Mapping : in Maps.Character_Mapping_Function);
57       -- String transformation subprograms
58          function Replace_Slice (Source   : in Bounded_String;
                                    Low      : in Positive;
                                    High     : in Natural;
                                    By       : in String;
                                    Drop     : in Truncation := Error)
             return Bounded_String;
59          procedure Replace_Slice (Source   : in out Bounded_String;
                                     Low      : in Positive;
                                     High     : in Natural;
                                     By       : in String;
                                     Drop     : in Truncation := Error);
60          function Insert (Source   : in Bounded_String;
                             Before   : in Positive;
                             New_Item : in String;
                             Drop     : in Truncation := Error)
             return Bounded_String;
61          procedure Insert (Source   : in out Bounded_String;
                              Before   : in Positive;
                              New_Item : in String;
                              Drop     : in Truncation := Error);
62          function Overwrite (Source    : in Bounded_String;
                                Position  : in Positive;
                                New_Item  : in String;
                                Drop      : in Truncation := Error)
             return Bounded_String;
63          procedure Overwrite (Source    : in out Bounded_String;
                                 Position  : in Positive;
                                 New_Item  : in String;
                                 Drop      : in Truncation := Error);
64          function Delete (Source  : in Bounded_String;
                             From    : in Positive;
                             Through : in Natural)
             return Bounded_String;
65          procedure Delete (Source  : in out Bounded_String;
                              From    : in Positive;
                              Through : in Natural);
66       --String selector subprograms
67          function Trim (Source : in Bounded_String;
                           Side   : in Trim_End)
             return Bounded_String;
          procedure Trim (Source : in out Bounded_String;
                          Side   : in Trim_End);
```

```
      function Trim (Source : in Bounded_String;                                    68
                     Left   : in Maps.Character_Set;
                     Right  : in Maps.Character_Set)
         return Bounded_String;

      procedure Trim (Source : in out Bounded_String;                               69
                      Left   : in Maps.Character_Set;
                      Right  : in Maps.Character_Set);

      function Head (Source : in Bounded_String;                                    70
                     Count  : in Natural;
                     Pad    : in Character  := Space;
                     Drop   : in Truncation := Error)
         return Bounded_String;

      procedure Head (Source : in out Bounded_String;                               71
                      Count  : in Natural;
                      Pad    : in Character  := Space;
                      Drop   : in Truncation := Error);

      function Tail (Source : in Bounded_String;                                    72
                     Count  : in Natural;
                     Pad    : in Character  := Space;
                     Drop   : in Truncation := Error)
         return Bounded_String;

      procedure Tail (Source : in out Bounded_String;                               73
                      Count  : in Natural;
                      Pad    : in Character  := Space;
                      Drop   : in Truncation := Error);

   --String constructor subprograms                                                  74

      function "*" (Left  : in Natural;                                             75
                    Right : in Character)
         return Bounded_String;

      function "*" (Left  : in Natural;                                             76
                    Right : in String)
         return Bounded_String;

      function "*" (Left  : in Natural;                                             77
                    Right : in Bounded_String)
         return Bounded_String;

      function Replicate (Count : in Natural;                                       78
                          Item  : in Character;
                          Drop  : in Truncation := Error)
         return Bounded_String;

      function Replicate (Count : in Natural;                                       79
                          Item  : in String;
                          Drop  : in Truncation := Error)
         return Bounded_String;

      function Replicate (Count : in Natural;                                       80
                          Item  : in Bounded_String;
                          Drop  : in Truncation := Error)
         return Bounded_String;

   private                                                                          81
      ... -- not specified by the language
   end Generic_Bounded_Length;

end Ada.Strings.Bounded;                                                            82
```

This paragraph was deleted.{*8652/0097*} {*AI95-00115-01*} {*AI95-00344-01*} 82.a.1/2

Null_Bounded_String represents the null string. If an object of type Bounded_String is not otherwise initialized, it will be initialized to the same value as Null_Bounded_String. 83

```
function Length (Source : in Bounded_String) return Length_Range;                  84
```

The Length function returns the length of the string represented by Source. 85

86
```
function To_Bounded_String (Source : in String;
                            Drop   : in Truncation := Error)
   return Bounded_String;
```

87/3 {*AI05-0264-1*} If Source'Length <= Max_Length, then this function returns a Bounded_String that represents Source. Otherwise, the effect depends on the value of Drop:

88 • If Drop=Left, then the result is a Bounded_String that represents the string comprising the rightmost Max_Length characters of Source.

89 • If Drop=Right, then the result is a Bounded_String that represents the string comprising the leftmost Max_Length characters of Source.

90 • If Drop=Error, then Strings.Length_Error is propagated.

91
```
function To_String (Source : in Bounded_String) return String;
```

92 To_String returns the String value with lower bound 1 represented by Source. If B is a Bounded_String, then B = To_Bounded_String(To_String(B)).

92.1/2
```
procedure Set_Bounded_String
   (Target :    out Bounded_String;
    Source : in     String;
    Drop   : in     Truncation := Error);
```

92.2/2 {*AI95-00301-01*} Equivalent to Target := To_Bounded_String (Source, Drop);

93 Each of the Append functions returns a Bounded_String obtained by concatenating the string or character given or represented by one of the parameters, with the string or character given or represented by the other parameter, and applying To_Bounded_String to the concatenation result string, with Drop as provided to the Append function.

94 Each of the procedures Append(Source, New_Item, Drop) has the same effect as the corresponding assignment Source := Append(Source, New_Item, Drop).

95 Each of the "&" functions has the same effect as the corresponding Append function, with Error as the Drop parameter.

96
```
function Element (Source : in Bounded_String;
                  Index  : in Positive)
   return Character;
```

97 Returns the character at position Index in the string represented by Source; propagates Index_Error if Index > Length(Source).

98
```
procedure Replace_Element (Source : in out Bounded_String;
                           Index  : in Positive;
                           By     : in Character);
```

99 Updates Source such that the character at position Index in the string represented by Source is By; propagates Index_Error if Index > Length(Source).

100
```
function Slice (Source : in Bounded_String;
                Low    : in Positive;
                High   : in Natural)
   return String;
```

101/1 {*8652/0049*} {*AI95-00128-01*} {*AI95-00238-01*} Returns the slice at positions Low through High in the string represented by Source; propagates Index_Error if Low > Length(Source)+1 or High > Length(Source). The bounds of the returned string are Low and High.

```
function Bounded_Slice                                                    101.1/2
   (Source : in Bounded_String;
    Low    : in Positive;
    High   : in Natural)
      return Bounded_String;
```

{*AI95-00301-01*} Returns the slice at positions Low through High in the string represented by Source as a bounded string; propagates Index_Error if Low > Length(Source)+1 or High > Length(Source). 101.2/2

```
procedure Bounded_Slice                                                   101.3/2
   (Source : in     Bounded_String;
    Target :    out Bounded_String;
    Low    : in     Positive;
    High   : in     Natural);
```

{*AI95-00301-01*} Equivalent to Target := Bounded_Slice (Source, Low, High); 101.4/2

Each of the functions "=", "<", ">", "<=", and ">=" returns the same result as the corresponding String operation applied to the String values given or represented by the two parameters. 102

Each of the search subprograms (Index, Index_Non_Blank, Count, Find_Token) has the same effect as the corresponding subprogram in Strings.Fixed applied to the string represented by the Bounded_String parameter. 103

Each of the Translate subprograms, when applied to a Bounded_String, has an analogous effect to the corresponding subprogram in Strings.Fixed. For the Translate function, the translation is applied to the string represented by the Bounded_String parameter, and the result is converted (via To_Bounded_String) to a Bounded_String. For the Translate procedure, the string represented by the Bounded_String parameter after the translation is given by the Translate function for fixed-length strings applied to the string represented by the original value of the parameter. 104

{*8652/0049*} {*AI95-00128-01*} Each of the transformation subprograms (Replace_Slice, Insert, Overwrite, Delete), selector subprograms (Trim, Head, Tail), and constructor functions ("*") has an effect based on its corresponding subprogram in Strings.Fixed, and Replicate is based on Fixed."*". In the case of a function, the corresponding fixed-length string subprogram is applied to the string represented by the Bounded_String parameter. To_Bounded_String is applied the result string, with Drop (or Error in the case of Generic_Bounded_Length."*") determining the effect when the string length exceeds Max_Length. In the case of a procedure, the corresponding function in Strings.Bounded.Generic_-Bounded_Length is applied, with the result assigned into the Source parameter. 105/1

Ramification: {*AI95-00114-01*} The "/=" operations between Bounded_String and String, and between String and Bounded_String, are automatically defined based on the corresponding "=" operations. 105.a/2

Implementation Advice

Bounded string objects should not be implemented by implicit pointers and dynamic allocation. 106

Implementation Advice: Bounded string objects should not be implemented by implicit pointers and dynamic allocation. 106.a.1/2

Implementation Note: The following is a possible implementation of the private part of the package: 106.a

```
type Bounded_String_Internals (Length : Length_Range := 0) is             106.b
   record
      Data : String(1..Length);
   end record;

type Bounded_String is                                                    106.c
   record
      Data : Bounded_String_Internals;  -- Unconstrained
   end record;
```

106.d
```
Null_Bounded_String : constant Bounded_String :=
   (Data => (Length => 0,
             Data   => (1..0 => ' ')));
```

Inconsistencies With Ada 95

106.e/2 {*AI95-00238-01*} **Amendment Correction:** The bounds of the string returned from Slice are now defined. This is technically an inconsistency; if a program depended on some other lower bound for the string returned from Slice, it could fail when compiled with Ada 2005. Such code is not portable even between Ada 95 implementations, so it should be very rare.

Incompatibilities With Ada 95

106.f/3 {*AI95-00301-01*} {*AI05-0005-1*} Procedure Set_Bounded_String, two Bounded_Slice subprograms, and overloaded versions of Index and Index_Non_Blank are added to Strings.Bounded.Generic_Bounded_Length. If an instance of Generic_Bounded_Length is referenced in a use_clause, and an entity *E* with the defining_identifier as a new entity in Generic_Bounded_Length is defined in a package that is also referenced in a use_clause, the entity *E* may no longer be use-visible, resulting in errors. This should be rare and is easily fixed if it does occur.

Wording Changes from Ada 95

106.g/2 {*8652/0049*} {*AI95-00128-01*} **Corrigendum:** Corrected the conditions for which Slice raises Index_Error.

106.h/2 {*8652/0049*} {*AI95-00128-01*} **Corrigendum:** Clarified the meaning of transformation, selector, and constructor subprograms by describing the effects of procedures and functions separately.

Incompatibilities With Ada 2005

106.i/3 {*AI05-0031-1*} An overloaded version of Find_Token is added to Strings.Bounded.Generic_Bounded_Length. If an instance of Generic_Bounded_Length is referenced in a use_clause, and an entity *E* with a defining_identifier of Find_Token is defined in a package that is also referenced in a use_clause, the entity *E* may no longer be use-visible, resulting in errors. This should be rare and is easily fixed if it does occur.

A.4.5 Unbounded-Length String Handling

1/5 {*AI12-0445-1*} The language-defined package Strings.Unbounded provides a private type Unbounded_String and a set of operations. An object of type Unbounded_String represents a String whose low bound is 1 and whose length can vary conceptually between 0 and Natural'Last. The subprograms for fixed-length string handling are either overloaded directly for Unbounded_String, or are modified as necessary to reflect the flexibility in length. Since the Unbounded_String type is private, relevant constructor and selector operations are provided.

1.a **Reason:** The transformation operations for fixed- and bounded-length strings that are not necessarily length preserving are supplied for Unbounded_String as procedures as well as functions. This allows an implementation to do an initial allocation for an unbounded string and to avoid further allocations as long as the length does not exceed the allocated length.

Static Semantics

2 The library package Strings.Unbounded has the following declaration:

3/5
```
{AI12-0241-1} {AI12-0302-1} with Ada.Strings.Maps;
package Ada.Strings.Unbounded
   with Preelaborate, Nonblocking, Global => in out synchronized is
```

4/5
```
   {AI95-00161-01} {AI12-0399-1}    type Unbounded_String is private
         with Preelaborable_Initialization;
```

5
```
   Null_Unbounded_String : constant Unbounded_String;
```

6
```
   function Length (Source : in Unbounded_String) return Natural;
```

7
```
   type String_Access is access all String;
   procedure Free (X : in out String_Access);
```

8
```
   -- Conversion, Concatenation, and Selection functions
```

9
```
   function To_Unbounded_String (Source : in String)
      return Unbounded_String;
```

```
   function To_Unbounded_String (Length : in Natural)                                  10
      return Unbounded_String;

   function To_String (Source : in Unbounded_String) return String;                   11

{AI95-00301-01}    procedure Set_Unbounded_String                                     11.1/2
     (Target :    out Unbounded_String;
      Source : in     String);

   procedure Append (Source   : in out Unbounded_String;                               12
                     New_Item : in Unbounded_String);

   procedure Append (Source   : in out Unbounded_String;                               13
                     New_Item : in String);

   procedure Append (Source   : in out Unbounded_String;                               14
                     New_Item : in Character);

   function "&" (Left, Right : in Unbounded_String)                                    15
      return Unbounded_String;

   function "&" (Left : in Unbounded_String; Right : in String)                        16
      return Unbounded_String;

   function "&" (Left : in String; Right : in Unbounded_String)                        17
      return Unbounded_String;

   function "&" (Left : in Unbounded_String; Right : in Character)                     18
      return Unbounded_String;

   function "&" (Left : in Character; Right : in Unbounded_String)                     19
      return Unbounded_String;

   function Element (Source : in Unbounded_String;                                     20
                     Index  : in Positive)
      return Character;

   procedure Replace_Element (Source : in out Unbounded_String;                        21
                              Index  : in Positive;
                              By     : in Character);

   function Slice (Source : in Unbounded_String;                                       22
                   Low    : in Positive;
                   High   : in Natural)
      return String;

{AI95-00301-01}    function Unbounded_Slice                                           22.1/2
      (Source : in Unbounded_String;
       Low    : in Positive;
       High   : in Natural)
          return Unbounded_String;

{AI95-00301-01}    procedure Unbounded_Slice                                          22.2/2
      (Source : in     Unbounded_String;
       Target :    out Unbounded_String;
       Low    : in     Positive;
       High   : in     Natural);

   function "="  (Left, Right : in Unbounded_String) return Boolean;                   23

   function "="  (Left : in Unbounded_String; Right : in String)                       24
     return Boolean;

   function "="  (Left : in String; Right : in Unbounded_String)                       25
     return Boolean;

   function "<"  (Left, Right : in Unbounded_String) return Boolean;                   26

   function "<"  (Left : in Unbounded_String; Right : in String)                       27
     return Boolean;

   function "<"  (Left : in String; Right : in Unbounded_String)                       28
     return Boolean;

   function "<=" (Left, Right : in Unbounded_String) return Boolean;                   29

   function "<="  (Left : in Unbounded_String; Right : in String)                      30
     return Boolean;
```

```
31        function "<="  (Left : in String; Right : in Unbounded_String)
            return Boolean;
32        function ">"  (Left, Right : in Unbounded_String) return Boolean;
33        function ">"  (Left : in Unbounded_String; Right : in String)
            return Boolean;
34        function ">"  (Left : in String; Right : in Unbounded_String)
            return Boolean;
35        function ">=" (Left, Right : in Unbounded_String) return Boolean;
36        function ">="  (Left : in Unbounded_String; Right : in String)
            return Boolean;
37        function ">="  (Left : in String; Right : in Unbounded_String)
            return Boolean;
38      -- Search subprograms
38.1/2  {AI95-00301-01}    function Index (Source  : in Unbounded_String;
                           Pattern : in String;
                           From    : in Positive;
                           Going   : in Direction := Forward;
                           Mapping : in Maps.Character_Mapping := Maps.Identity)
            return Natural;
38.2/2  {AI95-00301-01}    function Index (Source  : in Unbounded_String;
                           Pattern : in String;
                           From    : in Positive;
                           Going   : in Direction := Forward;
                           Mapping : in Maps.Character_Mapping_Function)
            return Natural;
39        function Index (Source   : in Unbounded_String;
                          Pattern  : in String;
                          Going    : in Direction := Forward;
                          Mapping  : in Maps.Character_Mapping
                                        := Maps.Identity)
            return Natural;
40        function Index (Source   : in Unbounded_String;
                          Pattern  : in String;
                          Going    : in Direction := Forward;
                          Mapping  : in Maps.Character_Mapping_Function)
            return Natural;
40.1/2  {AI95-00301-01}    function Index (Source  : in Unbounded_String;
                           Set     : in Maps.Character_Set;
                           From    : in Positive;
                           Test    : in Membership := Inside;
                           Going    : in Direction := Forward)
            return Natural;
41        function Index (Source : in Unbounded_String;
                          Set    : in Maps.Character_Set;
                          Test   : in Membership := Inside;
                          Going  : in Direction  := Forward) return Natural;
41.1/2  {AI95-00301-01}    function Index_Non_Blank (Source : in Unbounded_String;
                                  From   : in Positive;
                                  Going  : in Direction := Forward)
            return Natural;
42        function Index_Non_Blank (Source : in Unbounded_String;
                                  Going  : in Direction := Forward)
            return Natural;
43        function Count (Source   : in Unbounded_String;
                          Pattern  : in String;
                          Mapping  : in Maps.Character_Mapping
                                        := Maps.Identity)
            return Natural;
```

```
   function Count (Source   : in Unbounded_String;                                    44
                   Pattern  : in String;
                   Mapping  : in Maps.Character_Mapping_Function)
      return Natural;

   function Count (Source   : in Unbounded_String;                                    45
                   Set      : in Maps.Character_Set)
      return Natural;

{AI05-0031-1}    procedure Find_Token (Source : in Unbounded_String;                  45.1/3
                          Set    : in Maps.Character_Set;
                          From   : in Positive;
                          Test   : in Membership;
                          First  : out Positive;
                          Last   : out Natural);

   procedure Find_Token (Source : in Unbounded_String;                                46
                         Set    : in Maps.Character_Set;
                         Test   : in Membership;
                         First  : out Positive;
                         Last   : out Natural);

-- String translation subprograms                                                    47

   function Translate (Source  : in Unbounded_String;                                 48
                       Mapping : in Maps.Character_Mapping)
      return Unbounded_String;

   procedure Translate (Source  : in out Unbounded_String;                            49
                        Mapping : in Maps.Character_Mapping);

   function Translate (Source  : in Unbounded_String;                                 50
                       Mapping : in Maps.Character_Mapping_Function)
      return Unbounded_String;

   procedure Translate (Source  : in out Unbounded_String;                            51
                        Mapping : in Maps.Character_Mapping_Function);

-- String transformation subprograms                                                 52

   function Replace_Slice (Source   : in Unbounded_String;                            53
                           Low      : in Positive;
                           High     : in Natural;
                           By       : in String)
      return Unbounded_String;

   procedure Replace_Slice (Source   : in out Unbounded_String;                       54
                            Low      : in Positive;
                            High     : in Natural;
                            By       : in String);

   function Insert (Source   : in Unbounded_String;                                   55
                    Before   : in Positive;
                    New_Item : in String)
      return Unbounded_String;

   procedure Insert (Source   : in out Unbounded_String;                              56
                     Before   : in Positive;
                     New_Item : in String);

   function Overwrite (Source    : in Unbounded_String;                               57
                       Position  : in Positive;
                       New_Item  : in String)
      return Unbounded_String;

   procedure Overwrite (Source    : in out Unbounded_String;                          58
                        Position  : in Positive;
                        New_Item  : in String);

   function Delete (Source  : in Unbounded_String;                                    59
                    From    : in Positive;
                    Through : in Natural)
      return Unbounded_String;
```

```
60          procedure Delete (Source  : in out Unbounded_String;
                              From    : in Positive;
                              Through : in Natural);

61          function Trim (Source : in Unbounded_String;
                           Side   : in Trim_End)
               return Unbounded_String;

62          procedure Trim (Source : in out Unbounded_String;
                            Side   : in Trim_End);

63          function Trim (Source : in Unbounded_String;
                           Left   : in Maps.Character_Set;
                           Right  : in Maps.Character_Set)
               return Unbounded_String;

64          procedure Trim (Source : in out Unbounded_String;
                            Left   : in Maps.Character_Set;
                            Right  : in Maps.Character_Set);

65          function Head (Source : in Unbounded_String;
                           Count  : in Natural;
                           Pad    : in Character := Space)
               return Unbounded_String;

66          procedure Head (Source : in out Unbounded_String;
                            Count  : in Natural;
                            Pad    : in Character := Space);

67          function Tail (Source : in Unbounded_String;
                           Count  : in Natural;
                           Pad    : in Character := Space)
               return Unbounded_String;

68          procedure Tail (Source : in out Unbounded_String;
                            Count  : in Natural;
                            Pad    : in Character := Space);

69          function "*" (Left  : in Natural;
                          Right : in Character)
               return Unbounded_String;

70          function "*" (Left  : in Natural;
                          Right : in String)
               return Unbounded_String;

71          function "*" (Left  : in Natural;
                          Right : in Unbounded_String)
               return Unbounded_String;

72       private
            ... -- not specified by the language
         end Ada.Strings.Unbounded;
```

72.1/2 {*AI95-00360-01*} The type Unbounded_String needs finalization (see 7.6).

73 Null_Unbounded_String represents the null String. If an object of type Unbounded_String is not otherwise initialized, it will be initialized to the same value as Null_Unbounded_String.

74 The function Length returns the length of the String represented by Source.

75 The type String_Access provides a (nonprivate) access type for explicit processing of unbounded-length strings. The procedure Free performs an unchecked deallocation of an object of type String_Access.

76 The function To_Unbounded_String(Source : in String) returns an Unbounded_String that represents Source. The function To_Unbounded_String(Length : in Natural) returns an Unbounded_String that represents an uninitialized String whose length is Length.

77 The function To_String returns the String with lower bound 1 represented by Source. To_String and To_Unbounded_String are related as follows:

78 • If S is a String, then To_String(To_Unbounded_String(S)) = S.

- If U is an Unbounded_String, then To_Unbounded_String(To_String(U)) = U. 79

{*AI95-00301-01*} The procedure Set_Unbounded_String sets Target to an Unbounded_String that represents Source. 79.1/2

For each of the Append procedures, the resulting string represented by the Source parameter is given by the concatenation of the original value of Source and the value of New_Item. 80

Each of the "&" functions returns an Unbounded_String obtained by concatenating the string or character given or represented by one of the parameters, with the string or character given or represented by the other parameter, and applying To_Unbounded_String to the concatenation result string. 81

Ramification: {*AI12-0005-1*} If the resulting string is longer than Natural'Last, Constraint_Error is raised because the upper bound of the underlying String concatenation is outside of the range of the index subtype of Natural (see 4.5.3). Note that the same is true for other operations that attempt to create an overlong string: either they are defined in terms of this concatenation operation (as with Append) or they are defined in terms of Ada.Strings.Fixed operations (as with Insert and Replace_Slice), which themselves are defined in terms of String concatenation (which raises Constraint_Error for overlong strings as described above). Therefore, it is never possible to create an Unbounded_String with a length greater than Natural'Last. 81.a/5

The Element, Replace_Element, and Slice subprograms have the same effect as the corresponding bounded-length string subprograms. 82

{*AI95-00301-01*} {*AI05-0262-1*} The function Unbounded_Slice returns the slice at positions Low through High in the string represented by Source as an Unbounded_String. The procedure Unbounded_Slice sets Target to the Unbounded_String representing the slice at positions Low through High in the string represented by Source. Both subprograms propagate Index_Error if Low > Length(Source)+1 or High > Length(Source). 82.1/3

Each of the functions "=", "<", ">", "<=", and ">=" returns the same result as the corresponding String operation applied to the String values given or represented by Left and Right. 83

Each of the search subprograms (Index, Index_Non_Blank, Count, Find_Token) has the same effect as the corresponding subprogram in Strings.Fixed applied to the string represented by the Unbounded_String parameter. 84

The Translate function has an analogous effect to the corresponding subprogram in Strings.Fixed. The translation is applied to the string represented by the Unbounded_String parameter, and the result is converted (via To_Unbounded_String) to an Unbounded_String. 85

Each of the transformation functions (Replace_Slice, Insert, Overwrite, Delete), selector functions (Trim, Head, Tail), and constructor functions ("*") is likewise analogous to its corresponding subprogram in Strings.Fixed. For each of the subprograms, the corresponding fixed-length string subprogram is applied to the string represented by the Unbounded_String parameter, and To_Unbounded_String is applied the result string. 86

For each of the procedures Translate, Replace_Slice, Insert, Overwrite, Delete, Trim, Head, and Tail, the resulting string represented by the Source parameter is given by the corresponding function for fixed-length strings applied to the string represented by Source's original value. 87

Implementation Requirements

No storage associated with an Unbounded_String object shall be lost upon assignment or scope exit. 88

Implementation Note: {*AI95-00301-01*} A sample implementation of the private part of the package and several of the subprograms appears in the Ada 95 Rationale. 88.a/2

Incompatibilities With Ada 95

88.b/2 {*AI95-00360-01*} **Amendment Correction:** Type Unbounded_String is defined to need finalization. If the restriction No_Nested_Finalization (see D.7) applies to the partition, and Unbounded_String does not have a controlled part, it will not be allowed in local objects in Ada 2005 whereas it would be allowed in original Ada 95. Such code is not portable, as most Ada compilers have a controlled part in Unbounded_String, and thus would be illegal.

88.c/3 {*AI95-00301-01*} {*AI05-0005-1*} Procedure Set_Unbounded_String, two Unbounded_Slice subprograms, and overloaded versions of Index and Index_Non_Blank are added to Strings.Unbounded. If Strings.Unbounded is referenced in a use_clause, and an entity *E* with the same defining_identifier as a new entity in Strings.Unbounded is defined in a package that is also referenced in a use_clause, the entity *E* may no longer be use-visible, resulting in errors. This should be rare and is easily fixed if it does occur.

Extensions to Ada 95

88.d/2 {*AI95-00161-01*} **Amendment Correction:** Added a pragma Preelaborable_Initialization to type Unbounded_String, so that it can be used to declare default-initialized objects in preelaborated units.

Incompatibilities With Ada 2005

88.e/3 {*AI05-0031-1*} An overloaded version of Find_Token is added to Strings.Unbounded. If Strings.Unbounded is referenced in a use_clause, and an entity *E* with a defining_identifier of Find_Token is defined in a package that is also referenced in a use_clause, the entity *E* may no longer be use-visible, resulting in errors. This should be rare and is easily fixed if it does occur.

A.4.6 String-Handling Sets and Mappings

1 The language-defined package Strings.Maps.Constants declares Character_Set and Character_Mapping constants corresponding to classification and conversion functions in package Characters.Handling.

1.a **Discussion:** The Constants package is a child of Strings.Maps since it needs visibility of the private part of Strings.Maps in order to initialize the constants in a preelaborable way (i.e. via aggregates versus function calls).

Static Semantics

2 The library package Strings.Maps.Constants has the following declaration:

3/5
```
{AI95-00362-01} {AI12-0414-1} package Ada.Strings.Maps.Constants
   with Pure is

4     Control_Set           : constant Character_Set;
      Graphic_Set           : constant Character_Set;
      Letter_Set            : constant Character_Set;
      Lower_Set             : constant Character_Set;
      Upper_Set             : constant Character_Set;
      Basic_Set             : constant Character_Set;
      Decimal_Digit_Set     : constant Character_Set;
      Hexadecimal_Digit_Set : constant Character_Set;
      Alphanumeric_Set      : constant Character_Set;
      Special_Set           : constant Character_Set;
      ISO_646_Set           : constant Character_Set;

5     Lower_Case_Map        : constant Character_Mapping;
        --Maps to lower case for letters, else identity
      Upper_Case_Map        : constant Character_Mapping;
        --Maps to upper case for letters, else identity
      Basic_Map             : constant Character_Mapping;
        --Maps to basic letter for letters, else identity

6  private
      ... -- not specified by the language
   end Ada.Strings.Maps.Constants;
```

7 Each of these constants represents a correspondingly named set of characters or character mapping in Characters.Handling (see A.3.2).

8/5 NOTE {*AI05-0114-1*} {*AI12-0450-1*} There are certain characters which are defined to be lower case letters by ISO/IEC 10646 and are therefore allowed in identifiers, but are not considered lower case letters by Ada.Strings.Maps.Constants.

Reason: This is to maintain runtime compatibility with the Ada 95 definitions of these constants; existing correct programs could break if the definitions were changed in a way the programs did not anticipate. 8.a/3

Extensions to Ada 95

{*AI95-00362-01*} Strings.Maps.Constants is now Pure, so it can be used in pure units. 8.b/2

Wording Changes from Ada 2005

{*AI05-0114-1*} **Correction:** Added a note to clarify that these constants don't have any relationship to the characters allowed in identifiers. 8.c/3

A.4.7 Wide_String Handling

{*AI95-00302-03*} {*AI05-0286-1*} Facilities for handling strings of Wide_Character elements are found in the packages Strings.Wide_Maps, Strings.Wide_Fixed, Strings.Wide_Bounded, Strings.Wide_-Unbounded, and Strings.Wide_Maps.Wide_Constants, and in the library functions Strings.Wide_Hash, Strings.Wide_Fixed.Wide_Hash, Strings.Wide_Bounded.Wide_Hash, Strings.Wide_Unbounded.-Wide_Hash, Strings.Wide_Hash_Case_Insensitive, Strings.Wide_Fixed.Wide_Hash_Case_Insensitive, Strings.Wide_Bounded.Wide_Hash_Case_Insensitive, Strings.Wide_Unbounded.Wide_Hash_Case_-Insensitive, Strings.Wide_Equal_Case_Insensitive, Strings.Wide_Fixed.Wide_Equal_Case_Insensitive, Strings.Wide_Bounded.Wide_Equal_Case_Insensitive, and Strings.Wide_Unbounded.Wide_Equal_-Case_Insensitive. They provide the same string-handling operations as the corresponding packages and functions for strings of Character elements. 1/3

Static Semantics

The package Strings.Wide_Maps has the following declaration. 2

```
{AI12-0241-1} {AI12-0302-1} package Ada.Strings.Wide_Maps                                   3/5
   with Preelaborate, Nonblocking, Global => in out synchronized is
{AI95-00161-01} {AI12-0399-1}     -- Representation for a set of Wide_Character values:    4/5
   type Wide_Character_Set is private
      with Preelaborable_Initialization;
   Null_Set : constant Wide_Character_Set;                                                  5
   type Wide_Character_Range is                                                             6
     record
         Low  : Wide_Character;
         High : Wide_Character;
     end record;
   -- Represents Wide_Character range Low..High
   type Wide_Character_Ranges is array (Positive range <>)                                  7
      of Wide_Character_Range;
   function To_Set    (Ranges : in Wide_Character_Ranges)                                   8
      return Wide_Character_Set;
   function To_Set    (Span   : in Wide_Character_Range)                                    9
      return Wide_Character_Set;
   function To_Ranges (Set    : in Wide_Character_Set)                                      10
      return Wide_Character_Ranges;
   function "="   (Left, Right : in Wide_Character_Set) return Boolean;                     11
```

```
12          function "not" (Right : in Wide_Character_Set)
               return Wide_Character_Set;
            function "and" (Left, Right : in Wide_Character_Set)
               return Wide_Character_Set;
            function "or"  (Left, Right : in Wide_Character_Set)
               return Wide_Character_Set;
            function "xor" (Left, Right : in Wide_Character_Set)
               return Wide_Character_Set;
            function "-"   (Left, Right : in Wide_Character_Set)
               return Wide_Character_Set;
13          function Is_In (Element : in Wide_Character;
                            Set     : in Wide_Character_Set)
               return Boolean;
14          function Is_Subset (Elements : in Wide_Character_Set;
                                Set      : in Wide_Character_Set)
               return Boolean;
15          function "<=" (Left  : in Wide_Character_Set;
                           Right : in Wide_Character_Set)
               return Boolean renames Is_Subset;
16          -- Alternative representation for a set of Wide_Character values:
            subtype Wide_Character_Sequence is Wide_String;
17          function To_Set (Sequence  : in Wide_Character_Sequence)
               return Wide_Character_Set;
18          function To_Set (Singleton : in Wide_Character)
               return Wide_Character_Set;
19          function To_Sequence (Set  : in Wide_Character_Set)
               return Wide_Character_Sequence;
20/5     {AI95-00161-01} {AI12-0399-1}     -- Representation for a Wide_Character to Wide_Character mapping:
            type Wide_Character_Mapping is private
                with Preelaborable_Initialization;
21          function Value (Map     : in Wide_Character_Mapping;
                            Element : in Wide_Character)
               return Wide_Character;
22          Identity : constant Wide_Character_Mapping;
23          function To_Mapping (From, To : in Wide_Character_Sequence)
               return Wide_Character_Mapping;
24          function To_Domain (Map : in Wide_Character_Mapping)
               return Wide_Character_Sequence;
25          function To_Range  (Map : in Wide_Character_Mapping)
               return Wide_Character_Sequence;
26          type Wide_Character_Mapping_Function is
               access function (From : in Wide_Character) return Wide_Character;
27       private
            ... -- not specified by the language
         end Ada.Strings.Wide_Maps;
```

28 The context clause for each of the packages Strings.Wide_Fixed, Strings.Wide_Bounded, and Strings.Wide_Unbounded identifies Strings.Wide_Maps instead of Strings.Maps.

28.1/3 {*AI05-0223-1*} Types Wide_Character_Set and Wide_Character_Mapping need finalization.

29/3 {*AI95-00302-03*} {*AI05-0286-1*} For each of the packages Strings.Fixed, Strings.Bounded, Strings.Unbounded, and Strings.Maps.Constants, and for library functions Strings.Hash, Strings.Fixed.Hash, Strings.Bounded.Hash, Strings.Unbounded.Hash, Strings.Hash_Case_Insensitive, Strings.Fixed.Hash_Case_Insensitive, Strings.Bounded.Hash_Case_Insensitive, Strings.-Unbounded.Hash_Case_Insensitive, Strings.Equal_Case_Insensitive, Strings.Fixed.Equal_Case_-

Insensitive, Strings.Bounded.Equal_Case_Insensitive, and Strings.Unbounded.Equal_Case_Insensitive, the corresponding wide string package or function has the same contents except that

- Wide_Space replaces Space 30
- Wide_Character replaces Character 31
- Wide_String replaces String 32
- Wide_Character_Set replaces Character_Set 33
- Wide_Character_Mapping replaces Character_Mapping 34
- Wide_Character_Mapping_Function replaces Character_Mapping_Function 35
- Wide_Maps replaces Maps 36
- Bounded_Wide_String replaces Bounded_String 37
- Null_Bounded_Wide_String replaces Null_Bounded_String 38
- To_Bounded_Wide_String replaces To_Bounded_String 39
- To_Wide_String replaces To_String 40
- {*AI95-00301-01*} Set_Bounded_Wide_String replaces Set_Bounded_String 40.1/2
- Unbounded_Wide_String replaces Unbounded_String 41
- Null_Unbounded_Wide_String replaces Null_Unbounded_String 42
- Wide_String_Access replaces String_Access 43
- To_Unbounded_Wide_String replaces To_Unbounded_String 44
- {*AI95-00301-01*} Set_Unbounded_Wide_String replaces Set_Unbounded_String 44.1/2

The following additional declaration is present in Strings.Wide_Maps.Wide_Constants: 45

```
{AI95-00285-01} {AI95-00395-01} Character_Set : constant
Wide_Maps.Wide_Character_Set;
--Contains each Wide_Character value WC such that
--Characters.Conversions.Is_Character(WC) is True
```
46/2

{*AI95-00395-01*} Each Wide_Character_Set constant in the package Strings.Wide_Maps.Wide_Constants contains no values outside the Character portion of Wide_Character. Similarly, each Wide_Character_Mapping constant in this package is the identity mapping when applied to any element outside the Character portion of Wide_Character. 46.1/2

{*AI95-00362-01*} {*AI12-0302-1*} {*AI12-0414-1*} Aspect Pure is replaced by aspects `Preelaborate, Nonblocking, Global => in out synchronized` in Strings.Wide_Maps.Wide_Constants. 47/5

NOTE If a null Wide_Character_Mapping_Function is passed to any of the Wide_String handling subprograms, Constraint_Error is propagated. 48/5

Incompatibilities With Ada 95

{*AI95-00301-01*} Various new operations are added to Strings.Wide_Fixed, Strings.Wide_Bounded, and Strings.Wide_Unbounded. If one of these packages is referenced in a use_clause, and an entity *E* with the same defining_identifier as a new entity is defined in a package that is also referenced in a use_clause, the entity *E* may no longer be use-visible, resulting in errors. This should be rare and is easily fixed if it does occur. 49.a/2

Extensions to Ada 95

{*AI95-00161-01*} **Amendment Correction:** Added pragma Preelaborable_Initialization to types Wide_Character_Set and Wide_Character_Mapping, so that they can be used to declare default-initialized objects in preelaborated units. 49.b/2

Wording Changes from Ada 95

{*AI95-00285-01*} Corrected the description of Character_Set. 49.c/2

49.d/2 {*AI95-00302-03*} Added wide versions of Strings.Hash and Strings.Unbounded.Hash.

49.e/2 {*AI95-00362-01*} Added wording so that Strings.Wide_Maps.Wide_Constants does not change to Pure.

49.f/2 {*AI95-00395-01*} The second Note is now normative text, since there is no way to derive it from the other rules. It's a little weird given the use of Unicode character classifications in Ada 2005; but changing it would be inconsistent with Ada 95 and a one-to-one mapping isn't necessarily correct anyway.

Extensions to Ada 2005

49.g/3 {*AI05-0286-1*} The case insenstive library functions (Strings.Wide_Equal_Case_Insensitive, Strings.Wide_-Fixed.Wide_Equal_Case_Insensitive, Strings.Wide_Bounded.Wide_Equal_Case_Insensitive, Strings.Wide_-Unbounded.Wide_Equal_Case_Insensitive, Strings.Wide_Hash_Case_Insensitive, Strings.Wide_Fixed.Wide_Hash_-Case_Insensitive, Strings.Wide_Bounded.Wide_Hash_Case_Insensitive, and Strings.Wide_Unbounded.Wide_Hash_-Case_Insensitive) are new.

Wording Changes from Ada 2005

49.h/3 {*AI05-0223-1*} **Correction**: Identified Wide_Character_Set and Wide_Character_Mapping as needing finalization. It is likely that they are implemented with a controlled type, so this change is unlikely to make any difference in practice.

A.4.8 Wide_Wide_String Handling

1/3 {*AI95-00285-01*} {*AI95-00395-01*} {*AI05-0286-1*} Facilities for handling strings of Wide_Wide_Character elements are found in the packages Strings.Wide_Wide_Maps, Strings.-Wide_Wide_Fixed, Strings.Wide_Wide_Bounded, Strings.Wide_Wide_Unbounded, and Strings.-Wide_Wide_Maps.Wide_Wide_Constants, and in the library functions Strings.Wide_Wide_Hash, Strings.Wide_Wide_Fixed.Wide_Wide_Hash, Strings.Wide_Wide_Bounded.Wide_Wide_Hash, Strings.-Wide_Wide_Unbounded.Wide_Wide_Hash, Strings.Wide_Wide_Hash_Case_Insensitive, Strings.-Wide_Wide_Fixed.Wide_Wide_Hash_Case_Insensitive, Strings.Wide_Wide_Bounded.Wide_Wide_-Hash_Case_Insensitive, Strings.Wide_Wide_Unbounded.Wide_Wide_Hash_Case_Insensitive, Strings.-Wide_Wide_Equal_Case_Insensitive, Strings.Wide_Wide_Fixed.Wide_Wide_Equal_Case_Insensitive, Strings.Wide_Wide_Bounded.Wide_Wide_Equal_Case_Insensitive, and Strings.Wide_Wide_-Unbounded.Wide_Wide_Equal_Case_Insensitive. They provide the same string-handling operations as the corresponding packages and functions for strings of Character elements.

Static Semantics

2/2 {*AI95-00285-01*} The library package Strings.Wide_Wide_Maps has the following declaration.

```
3/5    {AI12-0241-1} {AI12-0302-1} package Ada.Strings.Wide_Wide_Maps
          with Preelaborate, Nonblocking, Global => in out synchronized is

4/5    {AI12-0399-1}     -- Representation for a set of Wide_Wide_Character values:
          type Wide_Wide_Character_Set is private
             with Preelaborable_Initialization;

5/2       Null_Set : constant Wide_Wide_Character_Set;

6/2       type Wide_Wide_Character_Range is
             record
                Low  : Wide_Wide_Character;
                High : Wide_Wide_Character;
             end record;
          -- Represents Wide_Wide_Character range Low..High

7/2       type Wide_Wide_Character_Ranges is array (Positive range <>)
                of Wide_Wide_Character_Range;

8/2       function To_Set (Ranges : in Wide_Wide_Character_Ranges)
                return Wide_Wide_Character_Set;

9/2       function To_Set (Span : in Wide_Wide_Character_Range)
                return Wide_Wide_Character_Set;

10/2      function To_Ranges (Set : in Wide_Wide_Character_Set)
                return Wide_Wide_Character_Ranges;
```

```
   function "=" (Left, Right : in Wide_Wide_Character_Set) return Boolean;                    11/2

   function "not" (Right : in Wide_Wide_Character_Set)                                        12/2
         return Wide_Wide_Character_Set;
   function "and" (Left, Right : in Wide_Wide_Character_Set)
         return Wide_Wide_Character_Set;
   function "or" (Left, Right : in Wide_Wide_Character_Set)
         return Wide_Wide_Character_Set;
   function "xor" (Left, Right : in Wide_Wide_Character_Set)
         return Wide_Wide_Character_Set;
   function "-" (Left, Right : in Wide_Wide_Character_Set)
         return Wide_Wide_Character_Set;

   function Is_In (Element : in Wide_Wide_Character;                                          13/2
                   Set     : in Wide_Wide_Character_Set)
         return Boolean;

   function Is_Subset (Elements : in Wide_Wide_Character_Set;                                 14/2
                       Set      : in Wide_Wide_Character_Set)
         return Boolean;

   function "<=" (Left  : in Wide_Wide_Character_Set;                                         15/2
                  Right : in Wide_Wide_Character_Set)
         return Boolean renames Is_Subset;

   -- Alternative representation for a set of Wide_Wide_Character values:                     16/2
   subtype Wide_Wide_Character_Sequence is Wide_Wide_String;

   function To_Set (Sequence : in Wide_Wide_Character_Sequence)                               17/2
         return Wide_Wide_Character_Set;

   function To_Set (Singleton : in Wide_Wide_Character)                                       18/2
         return Wide_Wide_Character_Set;

   function To_Sequence (Set : in Wide_Wide_Character_Set)                                    19/2
         return Wide_Wide_Character_Sequence;

{AI12-0399-1}     -- Representation for a Wide_Wide_Character to Wide_Wide_Character        20/5
   -- mapping:
   type Wide_Wide_Character_Mapping is private
      with Preelaborable_Initialization;

   function Value (Map     : in Wide_Wide_Character_Mapping;                                  21/2
                   Element : in Wide_Wide_Character)
         return Wide_Wide_Character;

   Identity : constant Wide_Wide_Character_Mapping;                                           22/2

   function To_Mapping (From, To : in Wide_Wide_Character_Sequence)                           23/2
         return Wide_Wide_Character_Mapping;

   function To_Domain (Map : in Wide_Wide_Character_Mapping)                                  24/2
         return Wide_Wide_Character_Sequence;

   function To_Range (Map : in Wide_Wide_Character_Mapping)                                   25/2
         return Wide_Wide_Character_Sequence;

   type Wide_Wide_Character_Mapping_Function is                                               26/2
         access function (From : in Wide_Wide_Character)
         return Wide_Wide_Character;

private                                                                                       27/2
   ... -- not specified by the language
end Ada.Strings.Wide_Wide_Maps;
```

{*AI95-00285-01*} The context clause for each of the packages Strings.Wide_Wide_Fixed, Strings.Wide_Wide_Bounded, and Strings.Wide_Wide_Unbounded identifies Strings.Wide_Wide_Maps instead of Strings.Maps. 28/2

{*AI05-0223-1*} Types Wide_Wide_Character_Set and Wide_Wide_Character_Mapping need finalization. 28.1/3

{*AI95-00285-01*} {*AI05-0286-1*} For each of the packages Strings.Fixed, Strings.Bounded, Strings.-Unbounded, and Strings.Maps.Constants, and for library functions Strings.Hash, Strings.Fixed.Hash, Strings.Bounded.Hash, Strings.Unbounded.Hash, Strings.Hash_Case_Insensitive, Strings.Fixed.Hash_- 29/3

Case_Insensitive, Strings.Bounded.Hash_Case_Insensitive, Strings.Unbounded.Hash_Case_Insensitive, Strings.Equal_Case_Insensitive, Strings.Fixed.Equal_Case_Insensitive, Strings.Bounded.Equal_Case_-Insensitive, and Strings.Unbounded.Equal_Case_Insensitive, the corresponding wide wide string package or function has the same contents except that

30/2 • Wide_Wide_Space replaces Space

31/2 • Wide_Wide_Character replaces Character

32/2 • Wide_Wide_String replaces String

33/2 • Wide_Wide_Character_Set replaces Character_Set

34/2 • Wide_Wide_Character_Mapping replaces Character_Mapping

35/2 • Wide_Wide_Character_Mapping_Function replaces Character_Mapping_Function

36/2 • Wide_Wide_Maps replaces Maps

37/2 • Bounded_Wide_Wide_String replaces Bounded_String

38/2 • Null_Bounded_Wide_Wide_String replaces Null_Bounded_String

39/2 • To_Bounded_Wide_Wide_String replaces To_Bounded_String

40/2 • To_Wide_Wide_String replaces To_String

41/2 • {*AI95-00301-01*} Set_Bounded_Wide_Wide_String replaces Set_Bounded_String

42/2 • Unbounded_Wide_Wide_String replaces Unbounded_String

43/2 • Null_Unbounded_Wide_Wide_String replaces Null_Unbounded_String

44/2 • Wide_Wide_String_Access replaces String_Access

45/2 • To_Unbounded_Wide_Wide_String replaces To_Unbounded_String

46/2 • {*AI95-00301-01*} Set_Unbounded_Wide_Wide_String replaces Set_Unbounded_String

47/2 {*AI95-00285-01*} {*AI95-00395-01*} The following additional declarations are present in Strings.Wide_Wide_Maps.Wide_Wide_Constants:

48/2
```
Character_Set : constant Wide_Wide_Maps.Wide_Wide_Character_Set;
-- Contains each Wide_Wide_Character value WWC such that
-- Characters.Conversions.Is_Character(WWC) is True
Wide_Character_Set : constant Wide_Wide_Maps.Wide_Wide_Character_Set;
-- Contains each Wide_Wide_Character value WWC such that
-- Characters.Conversions.Is_Wide_Character(WWC) is True
```

49/2 {*AI95-00395-01*} Each Wide_Wide_Character_Set constant in the package Strings.Wide_Wide_Maps.-Wide_Wide_Constants contains no values outside the Character portion of Wide_Wide_Character. Similarly, each Wide_Wide_Character_Mapping constant in this package is the identity mapping when applied to any element outside the Character portion of Wide_Wide_Character.

50/5 {*AI95-00395-01*} {*AI12-0302-1*} {*AI12-0414-1*} Aspect Pure is replaced by aspects `Preelaborate, Nonblocking, Global => in out synchronized` in Strings.Wide_Wide_Maps.Wide_Wide_Constants.

51/2 NOTE {*AI95-00285-01*} If a null Wide_Wide_Character_Mapping_Function is passed to any of the Wide_Wide_String handling subprograms, Constraint_Error is propagated.

Extensions to Ada 95

51.a/2 {*AI95-00285-01*} {*AI95-00395-01*} The double-wide string-handling packages (Strings.Wide_Wide_Maps, Strings.Wide_Wide_Fixed, Strings.Wide_Wide_Bounded, Strings.Wide_Wide_Unbounded, and Strings.Wide_Wide_Maps.Wide_Wide_Constants), and functions Strings.Wide_Wide_Hash and Strings.Wide_Wide_Unbounded.Wide_Wide_Hash are new.

Extensions to Ada 2005

{*AI05-0286-1*} The case insenstive library functions (Strings.Wide_Wide_Equal_Case_Insensitive, Strings.Wide_Wide_Fixed.Wide_Wide_Equal_Case_Insensitive, Strings.Wide_Wide_Bounded.Wide_Wide_Equal_-Case_Insensitive, Strings.Wide_Wide_Unbounded.Wide_Wide_Equal_Case_Insensitive, Strings.Wide_Wide_Hash_-Case_Insensitive, Strings.Wide_Wide_Fixed.Wide_Wide_Hash_Case_Insensitive, Strings.Wide_Wide_-Bounded.Wide_Wide_Hash_Case_Insensitive, and Strings.Wide_Wide_Unbounded.Wide_Wide_Hash_Case_-Insensitive) are new. 51.b/3

Wording Changes from Ada 2005

{*AI05-0223-1*} **Correction**: Identified Wide_Wide_Character_Set and Wide_Wide_Character_Mapping as needing finalization. It is likely that they are implemented with a controlled type, so this change is unlikely to make any difference in practice. 51.c/3

A.4.9 String Hashing

Static Semantics

{*AI95-00302-03*} The library function Strings.Hash has the following declaration: 1/2

```
{AI05-0298-1} {AI12-0414-1} with Ada.Containers;                                    2/5
function Ada.Strings.Hash (Key : String) return Containers.Hash_Type
   with Pure;
```

Returns an implementation-defined value which is a function of the value of Key. If *A* and *B* are strings such that *A* equals *B*, Hash(*A*) equals Hash(*B*). 3/2

Implementation defined: The values returned by Strings.Hash. 3.a/2

{*AI95-00302-03*} The library function Strings.Fixed.Hash has the following declaration: 4/2

```
{AI05-0298-1} with Ada.Containers, Ada.Strings.Hash;                                5/3
function Ada.Strings.Fixed.Hash (Key : String) return Containers.Hash_Type
   renames Ada.Strings.Hash;
```

{*AI95-00302-03*} The generic library function Strings.Bounded.Hash has the following declaration: 6/2

```
{AI05-0298-1} {AI12-0241-1} {AI12-0302-1} with Ada.Containers;                      7/5
generic
   with package Bounded is
      new Ada.Strings.Bounded.Generic_Bounded_Length (<>);
function Ada.Strings.Bounded.Hash (Key : Bounded.Bounded_String)
   return Containers.Hash_Type
   with Preelaborate, Nonblocking, Global => in out synchronized;
```

{*AI05-0001-1*} Equivalent to Strings.Hash (Bounded.To_String (Key)); 8/3

{*AI95-00302-03*} The library function Strings.Unbounded.Hash has the following declaration: 9/2

```
{AI05-0298-1} {AI12-0241-1} {AI12-0302-1} with Ada.Containers;                      10/5
function Ada.Strings.Unbounded.Hash (Key : Unbounded_String)
   return Containers.Hash_Type
   with Preelaborate, Nonblocking, Global => in out synchronized;
```

{*AI05-0001-1*} Equivalent to Strings.Hash (To_String (Key)); 11/3

{*AI05-0001-1*} {*AI05-0298-1*} The library function Strings.Hash_Case_Insensitive has the following declaration: 11.1/3

```
{AI12-0414-1} with Ada.Containers;                                                  11.2/5
function Ada.Strings.Hash_Case_Insensitive (Key : String)
   return Containers.Hash_Type
   with Pure;
```

Returns an implementation-defined value which is a function of the value of Key, converted to lower case. If A and B are strings such that Strings.Equal_Case_Insensitive (A, B) (see A.4.10) is True, then Hash_Case_Insensitive(A) equals Hash_Case_Insensitive(B). 11.3/3

11.4/3 {*AI05-0001-1*} {*AI05-0298-1*} The library function Strings.Fixed.Hash_Case_Insensitive has the following declaration:

11.5/3
```
with Ada.Containers, Ada.Strings.Hash_Case_Insensitive;
function Ada.Strings.Fixed.Hash_Case_Insensitive (Key : String)
   return Containers.Hash_Type renames Ada.Strings.Hash_Case_Insensitive;
```

11.6/3 {*AI05-0001-1*} {*AI05-0298-1*} The generic library function Strings.Bounded.Hash_Case_Insensitive has the following declaration:

11.7/5
```
{AI12-0241-1} {AI12-0302-1} with Ada.Containers;
generic
   with package Bounded is
      new Ada.Strings.Bounded.Generic_Bounded_Length (<>);
function Ada.Strings.Bounded.Hash_Case_Insensitive
   (Key : Bounded.Bounded_String) return Containers.Hash_Type
   with Preelaborate, Nonblocking, Global => in out synchronized;
```

11.8/3 Equivalent to Strings.Hash_Case_Insensitive (Bounded.To_String (Key));

11.9/3 {*AI05-0001-1*} {*AI05-0298-1*} The library function Strings.Unbounded.Hash_Case_Insensitive has the following declaration:

11.10/5
```
{AI12-0241-1} {AI12-0302-1} with Ada.Containers;
function Ada.Strings.Unbounded.Hash_Case_Insensitive
   (Key : Unbounded_String) return Containers.Hash_Type
   with Preelaborate, Nonblocking, Global => in out synchronized;
```

11.11/3 Equivalent to Strings.Hash_Case_Insensitive (To_String (Key));

Implementation Advice

12/2 {*AI95-00302-03*} The Hash functions should be good hash functions, returning a wide spread of values for different string values. It should be unlikely for similar strings to return the same value.

12.a/2 **Implementation Advice:** Strings.Hash should be good a hash function, returning a wide spread of values for different string values, and similar strings should rarely return the same value.

12.b/2 **Ramification:** The other functions are defined in terms of Strings.Hash, so they don't need separate advice in the Annex.

Extensions to Ada 95

12.c/2 {*AI95-00302-03*} The Strings.Hash, Strings.Fixed.Hash, Strings.Bounded.Hash, and Strings.Unbounded.Hash functions are new.

Extensions to Ada 2005

12.d/3 {*AI05-0001-1*} The Strings.Hash_Case_Insensitive, Strings.Fixed.Hash_Case_Insensitive, Strings.Bounded.Hash_Case_Insensitive, and Strings.Unbounded.Hash_Case_Insensitive functions are new.

A.4.10 String Comparison

Static Semantics

1/3 {*AI05-0001-1*} {*AI05-0286-1*} {*AI05-0298-1*} The library function Strings.Equal_Case_Insensitive has the following declaration:

2/5
```
{AI12-0414-1} function Ada.Strings.Equal_Case_Insensitive (Left, Right : String)
   return Boolean with Pure;
```

3/5 {*AI12-0263-1*} {*AI12-0450-1*} Returns True if the strings consist of the same sequence of characters after applying locale-independent simple case folding, as defined by documents referenced in Clause 2 of ISO/IEC 10646:2020. Otherwise, returns False. This function uses the same method as is used to determine whether two identifiers are the same.

Discussion: {*AI05-0286-1*} For String, this is equivalent to converting to lower case and comparing. Not so for other string types. For Wide_Strings and Wide_Wide_Strings, note that this result is a more accurate comparison than converting the strings to lower case and comparing the results; it is possible that the lower case conversions are the same but this routine will report the strings as different. Additionally, Unicode says that the result of this function will never change for strings made up solely of defined code points; there is no such guarantee for case conversion to lower case. 3.a/3

{*AI12-0263-1*} The "documents referenced" means Unicode, Chapter 4 (specifically, section 4.2 — Case). See the Implementation Notes in 2.3 for a source for machine-readable definitions of these properties. 3.b/5

{*AI05-0001-1*} {*AI05-0248-1*} {*AI05-0298-1*} The library function Strings.Fixed.Equal_Case_Insensitive has the following declaration: 4/3

```
with Ada.Strings.Equal_Case_Insensitive;                                    5/3
function Ada.Strings.Fixed.Equal_Case_Insensitive
   (Left, Right : String) return Boolean
      renames Ada.Strings.Equal_Case_Insensitive;
```

{*AI05-0001-1*} {*AI05-0248-1*} {*AI05-0298-1*} The generic library function Strings.Bounded.Equal_Case_Insensitive has the following declaration: 6/3

```
{AI12-0241-1} {AI12-0302-1} generic                                         7/5
   with package Bounded is
      new Ada.Strings.Bounded.Generic_Bounded_Length (<>);
function Ada.Strings.Bounded.Equal_Case_Insensitive
   (Left, Right : Bounded.Bounded_String) return Boolean
   with Preelaborate, Nonblocking, Global => in out synchronized;
```

Equivalent to Strings.Equal_Case_Insensitive (Bounded.To_String (Left), Bounded.To_String (Right)); 8/3

{*AI05-0001-1*} {*AI05-0248-1*} {*AI05-0298-1*} The library function Strings.Unbounded.Equal_Case_Insensitive has the following declaration: 9/3

```
{AI12-0241-1} {AI12-0302-1} function Ada.Strings.Unbounded.Equal_Case_Insensitive   10/5
   (Left, Right : Unbounded_String) return Boolean
   with Preelaborate, Nonblocking, Global => in out synchronized;
```

Equivalent to Strings.Equal_Case_Insensitive (To_String (Left), To_String (Right)); 11/3

{*AI05-0001-1*} {*AI05-0298-1*} The library function Strings.Less_Case_Insensitive has the following declaration: 12/3

```
{AI12-0414-1} function Ada.Strings.Less_Case_Insensitive (Left, Right : String)   13/5
   return Boolean with Pure;
```

Performs a lexicographic comparison of strings Left and Right, converted to lower case. 14/3

{*AI05-0001-1*} {*AI05-0248-1*} {*AI05-0298-1*} The library function Strings.Fixed.Less_Case_Insensitive has the following declaration: 15/3

```
with Ada.Strings.Less_Case_Insensitive;                                     16/3
function Ada.Strings.Fixed.Less_Case_Insensitive
   (Left, Right : String) return Boolean
      renames Ada.Strings.Less_Case_Insensitive;
```

{*AI05-0001-1*} {*AI05-0248-1*} {*AI05-0298-1*} The generic library function Strings.Bounded.Less_Case_Insensitive has the following declaration: 17/3

```
{AI12-0241-1} {AI12-0302-1} generic                                         18/5
   with package Bounded is
      new Ada.Strings.Bounded.Generic_Bounded_Length (<>);
function Ada.Strings.Bounded.Less_Case_Insensitive
  (Left, Right : Bounded.Bounded_String) return Boolean
   with Preelaborate, Nonblocking, Global => in out synchronized;
```

Equivalent to Strings.Less_Case_Insensitive (Bounded.To_String (Left), Bounded.To_String (Right)); 19/3

20/3 {*AI05-0001-1*} {*AI05-0248-1*} {*AI05-0298-1*} The library function Strings.Unbounded.Less_Case_Insensitive has the following declaration:

21/5
```
{AI12-0241-1} {AI12-0302-1} function Ada.Strings.Unbounded.Less_Case_Insensitive
  (Left, Right : Unbounded_String) return Boolean
   with Preelaborate, Nonblocking, Global => in out synchronized;
```

22/3 Equivalent to Strings.Less_Case_Insensitive (To_String (Left), To_String (Right));

Extensions to Ada 2005

22.a/3 {*AI05-0001-1*} {*AI05-0286-1*} The Strings.Equal_Case_Insensitive, Strings.Fixed.Equal_Case_Insensitive, Strings.Bounded.Equal_Case_Insensitive, Strings.Unbounded.Equal_Case_Insensitive, Strings.Less_Case_Insensitive, Strings.Fixed.Less_Case_Insensitive, Strings.Bounded.Less_Case_Insensitive, Strings.Unbounded.Less_Case_Insensitive functions are new.

A.4.11 String Encoding

1/3 {*AI05-0137-2*} Facilities for encoding, decoding, and converting strings in various character encoding schemes are provided by packages Strings.UTF_Encoding, Strings.UTF_Encoding.Conversions, Strings.UTF_Encoding.Strings, Strings.UTF_Encoding.Wide_Strings, and Strings.UTF_Encoding.Wide_Wide_Strings.

Static Semantics

2/3 {*AI05-0137-2*} The encoding library packages have the following declarations:

3/5
```
{AI05-0137-2} {AI12-0414-1} package Ada.Strings.UTF_Encoding
  with Pure is
```

4/3
```
   -- Declarations common to the string encoding packages
   type Encoding_Scheme is (UTF_8, UTF_16BE, UTF_16LE);
```

5/3
```
   subtype UTF_String is String;
```

6/3
```
   subtype UTF_8_String is String;
```

7/3
```
   subtype UTF_16_Wide_String is Wide_String;
```

8/3
```
   Encoding_Error : exception;
```

9/3
```
   BOM_8    : constant UTF_8_String :=
                 Character'Val(16#EF#) &
                 Character'Val(16#BB#) &
                 Character'Val(16#BF#);
```

10/3
```
   BOM_16BE : constant UTF_String :=
                 Character'Val(16#FE#) &
                 Character'Val(16#FF#);
```

11/3
```
   BOM_16LE : constant UTF_String :=
                 Character'Val(16#FF#) &
                 Character'Val(16#FE#);
```

12/3
```
   BOM_16   : constant UTF_16_Wide_String :=
                (1 => Wide_Character'Val(16#FEFF#));
```

13/3
```
   function Encoding (Item    : UTF_String;
                      Default : Encoding_Scheme := UTF_8)
      return Encoding_Scheme;
```

14/3
```
end Ada.Strings.UTF_Encoding;
```

15/5
```
{AI05-0137-2} {AI12-0414-1} package Ada.Strings.UTF_Encoding.Conversions
   with Pure is
```

16/3
```
   -- Conversions between various encoding schemes
   function Convert (Item          : UTF_String;
                     Input_Scheme  : Encoding_Scheme;
                     Output_Scheme : Encoding_Scheme;
                     Output_BOM    : Boolean := False) return UTF_String;
```

```
   function Convert (Item          : UTF_String;                                    17/3
                     Input_Scheme  : Encoding_Scheme;
                     Output_BOM    : Boolean := False)
      return UTF_16_Wide_String;

   function Convert (Item          : UTF_8_String;                                  18/3
                     Output_BOM    : Boolean := False)
      return UTF_16_Wide_String;

   function Convert (Item          : UTF_16_Wide_String;                            19/3
                     Output_Scheme : Encoding_Scheme;
                     Output_BOM    : Boolean := False) return UTF_String;

   function Convert (Item          : UTF_16_Wide_String;                            20/3
                     Output_BOM    : Boolean := False) return UTF_8_String;

end Ada.Strings.UTF_Encoding.Conversions;                                           21/3

{AI05-0137-2} {AI12-0414-1} package Ada.Strings.UTF_Encoding.Strings                22/5
   with Pure is

   -- Encoding / decoding between String and various encoding schemes               23/3
   function Encode (Item          : String;
                    Output_Scheme : Encoding_Scheme;
                    Output_BOM    : Boolean  := False) return UTF_String;

   function Encode (Item       : String;                                            24/3
                    Output_BOM : Boolean  := False) return UTF_8_String;

   function Encode (Item       : String;                                            25/3
                    Output_BOM : Boolean  := False)
      return UTF_16_Wide_String;

   function Decode (Item         : UTF_String;                                      26/3
                    Input_Scheme : Encoding_Scheme) return String;

   function Decode (Item : UTF_8_String) return String;                             27/3

   function Decode (Item : UTF_16_Wide_String) return String;                       28/3

end Ada.Strings.UTF_Encoding.Strings;                                               29/3

{AI05-0137-2} {AI12-0414-1} package Ada.Strings.UTF_Encoding.Wide_Strings           30/5
   with Pure is

   -- Encoding / decoding between Wide_String and various encoding schemes         31/3
   function Encode (Item          : Wide_String;
                    Output_Scheme : Encoding_Scheme;
                    Output_BOM    : Boolean  := False) return UTF_String;

   function Encode (Item       : Wide_String;                                       32/3
                    Output_BOM : Boolean  := False) return UTF_8_String;

   function Encode (Item       : Wide_String;                                       33/3
                    Output_BOM : Boolean  := False)
      return UTF_16_Wide_String;

   function Decode (Item         : UTF_String;                                      34/3
                    Input_Scheme : Encoding_Scheme) return Wide_String;

   function Decode (Item : UTF_8_String) return Wide_String;                        35/3

   function Decode (Item : UTF_16_Wide_String) return Wide_String;                  36/3

end Ada.Strings.UTF_Encoding.Wide_Strings;                                          37/3

{AI05-0137-2} {AI12-0414-1} package Ada.Strings.UTF_Encoding.Wide_Wide_Strings      38/5
   with Pure is

   -- Encoding / decoding between Wide_Wide_String and various encoding schemes    39/3
   function Encode (Item          : Wide_Wide_String;
                    Output_Scheme : Encoding_Scheme;
                    Output_BOM    : Boolean  := False) return UTF_String;

   function Encode (Item       : Wide_Wide_String;                                  40/3
                    Output_BOM : Boolean  := False) return UTF_8_String;
```

41/3
```
      function Encode (Item       : Wide_Wide_String;
                       Output_BOM : Boolean  := False)
         return UTF_16_Wide_String;
```

42/3
```
      function Decode (Item         : UTF_String;
                       Input_Scheme : Encoding_Scheme) return Wide_Wide_String;
```

43/3
```
      function Decode (Item : UTF_8_String) return Wide_Wide_String;
```

44/3
```
      function Decode (Item : UTF_16_Wide_String) return Wide_Wide_String;
```

45/3
```
   end Ada.Strings.UTF_Encoding.Wide_Wide_Strings;
```

46/3 {*AI05-0137-2*} {*AI05-0262-1*} The type Encoding_Scheme defines encoding schemes. UTF_8 corresponds to the UTF-8 encoding scheme defined by Annex D of ISO/IEC 10646. UTF_16BE corresponds to the UTF-16 encoding scheme defined by Annex C of ISO/IEC 10646 in 8 bit, big-endian order; and UTF_16LE corresponds to the UTF-16 encoding scheme in 8 bit, little-endian order.

47/3 {*AI05-0137-2*} The subtype UTF_String is used to represent a String of 8-bit values containing a sequence of values encoded in one of three ways (UTF-8, UTF-16BE, or UTF-16LE). The subtype UTF_8_String is used to represent a String of 8-bit values containing a sequence of values encoded in UTF-8. The subtype UTF_16_Wide_String is used to represent a Wide_String of 16-bit values containing a sequence of values encoded in UTF-16.

48/3 {*AI05-0137-2*} {*AI05-0262-1*} The BOM_8, BOM_16BE, BOM_16LE, and BOM_16 constants correspond to values used at the start of a string to indicate the encoding.

49/3 {*AI05-0262-1*} {*AI05-0269-1*} Each of the Encode functions takes a String, Wide_String, or Wide_Wide_String Item parameter that is assumed to be an array of unencoded characters. Each of the Convert functions takes a UTF_String, UTF_8_String, or UTF_16_String Item parameter that is assumed to contain characters whose position values correspond to a valid encoding sequence according to the encoding scheme required by the function or specified by its Input_Scheme parameter.

50/3 {*AI05-0137-2*} {*AI05-0262-1*} {*AI05-0269-1*} Each of the Convert and Encode functions returns a UTF_String, UTF_8_String, or UTF_16_String value whose characters have position values that correspond to the encoding of the Item parameter according to the encoding scheme required by the function or specified by its Output_Scheme parameter. For UTF_8, no overlong encoding is returned. A BOM is included at the start of the returned string if the Output_BOM parameter is set to True. The lower bound of the returned string is 1.

51/3 {*AI05-0137-2*} {*AI05-0262-1*} Each of the Decode functions takes a UTF_String, UTF_8_String, or UTF_16_String Item parameter which is assumed to contain characters whose position values correspond to a valid encoding sequence according to the encoding scheme required by the function or specified by its Input_Scheme parameter, and returns the corresponding String, Wide_String, or Wide_Wide_String value. The lower bound of the returned string is 1.

52/3 {*AI05-0137-2*} {*AI05-0262-1*} For each of the Convert and Decode functions, an initial BOM in the input that matches the expected encoding scheme is ignored, and a different initial BOM causes Encoding_Error to be propagated.

53/3 {*AI05-0137-2*} The exception Encoding_Error is also propagated in the following situations:

54/4 - {*AI12-0088-1*} By a Convert or Decode function when a UTF encoded string contains an invalid encoding sequence.

54.a/4 **To be honest:** {*AI12-0088-1*} An overlong encoding is not invalid for the purposes of this check, and this does not depend on the character set version in use. Some recent character set standards declare overlong encodings to be invalid; it would be unnecessary and unfriendly to users for Convert or Decode to raise an exception for an overlong encoding.

- {*AI12-0088-1*} By a Convert or Decode function when the expected encoding is UTF-16BE or UTF-16LE and the input string has an odd length. 55/4
- {*AI05-0262-1*} By a Decode function yielding a String when the decoding of a sequence results in a code point whose value exceeds 16#FF#. 56/3
- By a Decode function yielding a Wide_String when the decoding of a sequence results in a code point whose value exceeds 16#FFFF#. 57/3
- {*AI05-0262-1*} By an Encode function taking a Wide_String as input when an invalid character appears in the input. In particular, the characters whose position is in the range 16#D800# .. 16#DFFF# are invalid because they conflict with UTF-16 surrogate encodings, and the characters whose position is 16#FFFE# or 16#FFFF# are also invalid because they conflict with BOM codes. 58/3

{*AI05-0137-2*} 59/3
```
function Encoding (Item    : UTF_String;
                   Default : Encoding_Scheme := UTF_8)
   return Encoding_Scheme;
```

{*AI05-0137-2*} {*AI05-0269-1*} Inspects a UTF_String value to determine whether it starts with a BOM for UTF-8, UTF-16BE, or UTF_16LE. If so, returns the scheme corresponding to the BOM; otherwise, returns the value of Default. 60/3

{*AI05-0137-2*} 61/3
```
function Convert (Item          : UTF_String;
                  Input_Scheme  : Encoding_Scheme;
                  Output_Scheme : Encoding_Scheme;
                  Output_BOM    : Boolean := False) return UTF_String;
```

Returns the value of Item (originally encoded in UTF-8, UTF-16LE, or UTF-16BE as specified by Input_Scheme) encoded in one of these three schemes as specified by Output_Scheme. 62/3

{*AI05-0137-2*} 63/3
```
function Convert (Item          : UTF_String;
                  Input_Scheme  : Encoding_Scheme;
                  Output_BOM    : Boolean := False)
   return UTF_16_Wide_String;
```

Returns the value of Item (originally encoded in UTF-8, UTF-16LE, or UTF-16BE as specified by Input_Scheme) encoded in UTF-16. 64/3

{*AI05-0137-2*} 65/3
```
function Convert (Item          : UTF_8_String;
                  Output_BOM    : Boolean := False)
   return UTF_16_Wide_String;
```

Returns the value of Item (originally encoded in UTF-8) encoded in UTF-16. 66/3

{*AI05-0137-2*} 67/3
```
function Convert (Item          : UTF_16_Wide_String;
                  Output_Scheme : Encoding_Scheme;
                  Output_BOM    : Boolean := False) return UTF_String;
```

Returns the value of Item (originally encoded in UTF-16) encoded in UTF-8, UTF-16LE, or UTF-16BE as specified by Output_Scheme. 68/3

{*AI05-0137-2*} 69/3
```
function Convert (Item          : UTF_16_Wide_String;
                  Output_BOM    : Boolean := False) return UTF_8_String;
```

Returns the value of Item (originally encoded in UTF-16) encoded in UTF-8. 70/3

{*AI05-0137-2*} 71/3
```
function Encode (Item          : String;
                 Output_Scheme : Encoding_Scheme;
                 Output_BOM    : Boolean  := False) return UTF_String;
```

{*AI05-0262-1*} Returns the value of Item encoded in UTF-8, UTF-16LE, or UTF-16BE as specified by Output_Scheme. 72/3

73/3 {*AI05-0137-2*} `function Encode (Item : String; Output_BOM : Boolean := False) return UTF_8_String;`

74/3 Returns the value of Item encoded in UTF-8.

75/3 {*AI05-0137-2*} `function Encode (Item : String; Output_BOM : Boolean := False) return UTF_16_Wide_String;`

76/3 Returns the value of Item encoded in UTF_16.

77/3 {*AI05-0137-2*} `function Decode (Item : UTF_String; Input_Scheme : Encoding_Scheme) return String;`

78/3 Returns the result of decoding Item, which is encoded in UTF-8, UTF-16LE, or UTF-16BE as specified by Input_Scheme.

79/3 {*AI05-0137-2*} `function Decode (Item : UTF_8_String) return String;`

80/3 Returns the result of decoding Item, which is encoded in UTF-8.

81/3 {*AI05-0137-2*} `function Decode (Item : UTF_16_Wide_String) return String;`

82/3 Returns the result of decoding Item, which is encoded in UTF-16.

83/3 {*AI05-0137-2*} `function Encode (Item : Wide_String; Output_Scheme : Encoding_Scheme; Output_BOM : Boolean := False) return UTF_String;`

84/3 {*AI05-0262-1*} Returns the value of Item encoded in UTF-8, UTF-16LE, or UTF-16BE as specified by Output_Scheme.

85/3 {*AI05-0137-2*} `function Encode (Item : Wide_String; Output_BOM : Boolean := False) return UTF_8_String;`

86/3 Returns the value of Item encoded in UTF-8.

87/3 {*AI05-0137-2*} `function Encode (Item : Wide_String; Output_BOM : Boolean := False) return UTF_16_Wide_String;`

88/3 Returns the value of Item encoded in UTF_16.

89/3 {*AI05-0137-2*} `function Decode (Item : UTF_String; Input_Scheme : Encoding_Scheme) return Wide_String;`

90/3 Returns the result of decoding Item, which is encoded in UTF-8, UTF-16LE, or UTF-16BE as specified by Input_Scheme.

91/3 {*AI05-0137-2*} `function Decode (Item : UTF_8_String) return Wide_String;`

92/3 Returns the result of decoding Item, which is encoded in UTF-8.

93/3 {*AI05-0137-2*} `function Decode (Item : UTF_16_Wide_String) return Wide_String;`

94/3 Returns the result of decoding Item, which is encoded in UTF-16.

95/3 {*AI05-0137-2*} `function Encode (Item : Wide_Wide_String; Output_Scheme : Encoding_Scheme; Output_BOM : Boolean := False) return UTF_String;`

96/3 {*AI05-0262-1*} Returns the value of Item encoded in UTF-8, UTF-16LE, or UTF-16BE as specified by Output_Scheme.

97/3 {*AI05-0137-2*} `function Encode (Item : Wide_Wide_String; Output_BOM : Boolean := False) return UTF_8_String;`

98/3 Returns the value of Item encoded in UTF-8.

{*AI05-0137-2*} **function** Encode (Item : Wide_Wide_String; 99/3
Output_BOM : Boolean := False) **return** UTF_16_Wide_String;

Returns the value of Item encoded in UTF_16. 100/3

{*AI05-0137-2*} **function** Decode (Item : UTF_String; 101/3
Input_Scheme : Encoding_Scheme) **return** Wide_Wide_String;

Returns the result of decoding Item, which is encoded in UTF-8, UTF-16LE, or UTF-16BE as specified by Input_Scheme. 102/3

{*AI05-0137-2*} **function** Decode (Item : UTF_8_String) **return** Wide_Wide_String; 103/3

Returns the result of decoding Item, which is encoded in UTF-8. 104/3

{*AI05-0137-2*} **function** Decode (Item : UTF_16_Wide_String) **return** Wide_Wide_String; 105/3

Returns the result of decoding Item, which is encoded in UTF-16. 106/3

Implementation Advice

{*AI05-0137-2*} If an implementation supports other encoding schemes, another similar child of Ada.Strings should be defined. 107/3

Implementation Advice: If an implementation supports other string encoding schemes, a child of Ada.Strings similar to UTF_Encoding should be defined. 107.a.1/3

NOTE {*AI05-0137-2*} A BOM (Byte-Order Mark, code position 16#FEFF#) can be included in a file or other entity to indicate the encoding; it is skipped when decoding. Typically, only the first line of a file or other entity contains a BOM. When decoding, the Encoding function can be called on the first line to determine the encoding; this encoding will then be used in subsequent calls to Decode to convert all of the lines to an internal format. 108/3

Extensions to Ada 2005

{*AI05-0137-2*} The packages Strings.UTF_Encoding, Strings.UTF_Encoding.Conversions, Strings.UTF_Encoding.Strings, Strings.UTF_Encoding.Wide_Strings, and Strings.UTF_Encoding.Wide_Wide_Strings are new. 108.a/3

Wording Changes from Ada 2012

{*AI12-0088-1*} **Corrigendum:** Fixed the omission that Convert routines make the same checks on input as Decode routines. 108.b/4

A.4.12 Universal Text Buffers

{*AI12-0340-1*} {*AI12-0439-1*} A universal text buffer can be used to save and retrieve text of any language-defined string type. The types used to save and retrieve the text can be different. 1/5

Static Semantics

{*AI12-0340-1*} The text buffer library packages have the following declarations: 2/5

```
{AI12-0340-1} {AI12-0384-2} with Ada.Strings.UTF_Encoding.Wide_Wide_Strings;
package Ada.Strings.Text_Buffers
   with Pure is

   type Text_Buffer_Count is range 0 .. implementation-defined;

   New_Line_Count : constant Text_Buffer_Count := implementation-defined;

   type Root_Buffer_Type is abstract tagged private
      with Default_Initial_Condition =>
             Current_Indent (Root_Buffer_Type) = 0;

   procedure Put (
      Buffer : in out Root_Buffer_Type;
      Item   : in     String) is abstract;
```

3/5 4/5 5/5 6/5 7/5

```
8/5         procedure Wide_Put (
               Buffer : in out Root_Buffer_Type;
               Item   : in     Wide_String) is abstract;
9/5         procedure Wide_Wide_Put (
               Buffer : in out Root_Buffer_Type;
               Item   : in     Wide_Wide_String) is abstract;
10/5        procedure Put_UTF_8 (
               Buffer : in out Root_Buffer_Type;
               Item   : in     UTF_Encoding.UTF_8_String) is abstract;
11/5        procedure Wide_Put_UTF_16 (
               Buffer : in out Root_Buffer_Type;
               Item   : in     UTF_Encoding.UTF_16_Wide_String) is abstract;
12/5        procedure New_Line (Buffer : in out Root_Buffer_Type) is abstract;
13/5        Standard_Indent : constant Text_Buffer_Count := 3;
14/5        function Current_Indent (
               Buffer : Root_Buffer_Type) return Text_Buffer_Count;
15/5        procedure Increase_Indent (
               Buffer : in out Root_Buffer_Type;
               Amount : in     Text_Buffer_Count := Standard_Indent)
               with Post'Class =>
                  Current_Indent (Buffer) = Current_Indent (Buffer)'Old + Amount;
16/5        procedure Decrease_Indent (
               Buffer : in out Root_Buffer_Type;
               Amount : in     Text_Buffer_Count := Standard_Indent)
               with Pre'Class =>
                       Current_Indent (Buffer) >= Amount
                          or else raise Constraint_Error,
                    Post'Class =>
                       Current_Indent (Buffer) =
                          Current_Indent (Buffer)'Old - Amount;
17/5     private
            ... -- not specified by the language
         end Ada.Strings.Text_Buffers;
18/5     {AI12-0340-1} {AI12-0384-2} package Ada.Strings.Text_Buffers.Unbounded
            with Preelaborate, Nonblocking, Global => null is
19/5        type Buffer_Type is new Root_Buffer_Type with private;
20/5        function Get (
               Buffer : in out Buffer_Type)
               return String
               with Post'Class =>
                  Get'Result'First = 1 and then Current_Indent (Buffer) = 0;
21/5        function Wide_Get (
               Buffer : in out Buffer_Type)
               return Wide_String
               with Post'Class =>
                  Wide_Get'Result'First = 1 and then Current_Indent (Buffer) = 0;
22/5        function Wide_Wide_Get (
               Buffer : in out Buffer_Type)
               return Wide_Wide_String
               with Post'Class =>
                  Wide_Wide_Get'Result'First = 1
                     and then Current_Indent (Buffer) = 0;
23/5        function Get_UTF_8 (
               Buffer : in out Buffer_Type)
               return UTF_Encoding.UTF_8_String
               with Post'Class =>
                  Get_UTF_8'Result'First = 1 and then Current_Indent (Buffer) = 0;
```

```
   function Wide_Get_UTF_16 (                                                    24/5
      Buffer : in out Buffer_Type)
      return UTF_Encoding.UTF_16_Wide_String
      with Post'Class =>
         Wide_Get_UTF_16'Result'First = 1
            and then Current_Indent (Buffer) = 0;

private                                                                          25/5
   ... -- not specified by the language, but will include nonabstract
       -- overridings of all inherited subprograms that require overriding.
end Ada.Strings.Text_Buffers.Unbounded;

{AI12-0340-1} {AI12-0384-2} package Ada.Strings.Text_Buffers.Bounded             26/5
   with Pure, Nonblocking, Global => null is

   type Buffer_Type (Max_Characters : Text_Buffer_Count)                         27/5
      is new Root_Buffer_Type with private
      with Default_Initial_Condition => not Text_Truncated (Buffer_Type);

   function Text_Truncated (Buffer : in Buffer_Type) return Boolean;             28/5

   -- Get, Wide_Get, Wide_Wide_Get, Get_UTF_8, and Wide_Get_UTF_16               29/5
   -- are declared here just as in the Unbounded child.

private                                                                          30/5
   ... -- not specified by the language, but will include nonabstract
       -- overridings of all inherited subprograms that require overriding.
end Ada.Strings.Text_Buffers.Bounded;
```

{*AI12-0340-1*} Character_Count returns the number of characters currently stored in a text buffer. 31/5

Ramification: {*AI12-0340-1*} This is lower-case "characters". The representation isn't considered, so it is irrelevant what type of character (Character, Wide_Character, or Wide_Wide_Character) was stored. 31.a/5

{*AI12-0340-1*} {*AI12-0384-2*} New_Line stores New_Line_Count characters that represent a new line into a text buffer. Current_Indent returns the current indentation associated with the buffer, with zero meaning there is no indentation in effect; Increase_Indent and Decrease_Indent increase or decrease the indentation associated with the buffer. 32/5

{*AI12-0340-1*} {*AI12-0384-2*} A call to Put, Wide_Put, Wide_Wide_Put, Put_UTF_8, or Wide_Put_UTF_16 stores a sequence of characters into the text buffer, preceded by Current_Indent(Buffer) spaces (Wide_Wide_Characters with position 32) if there is at least one character in Item and it would have been the first character on the current line. 33/5

{*AI12-0340-1*} {*AI12-0384-2*} A call to function Get, Wide_Get, Wide_Wide_Get, Get_UTF_8, or Wide_Get_UTF_16 returns the same sequence of characters as was present in the calls that stored the characters into the buffer, if representable. For a call to Get, if any character in the sequence is not defined in Character, the result is implementation defined. Similarly, for a call to Wide_Get, if any character in the sequence is not defined in Wide_Character, the result is implementation defined. As part of a call on any of the Get functions, the buffer is reset to an empty state, with no stored characters. 34/5

Implementation defined: The value returned by a call to a Text_Buffer Get procedure if any character in the returned sequence is not defined in Character. 34.a/5

Implementation defined: The value returned by a call to a Text_Buffer Wide_Get procedure if any character in the returned sequence is not defined in Wide_Character. 34.b/5

{*AI12-0384-2*} In the case of a Buf of type Text_Buffers.Bounded.Buffer_Type, Text_Truncated (Buf) returns True if the various Put procedures together have attempted to store more than Buf.Max_Characters into Buf. If this function returns True, then the various Get functions return a representation of only the first Buf.Max_Characters characters that were stored in Buf. 35/5

Implementation Advice

{*AI12-0340-1*} Bounded buffer objects should be implemented without dynamic allocation. 36/5

Implementation Advice: Bounded buffer objects should be implemented without dynamic allocation. 36.a.1/5

Extensions to Ada 2012

36.a/5 {*AI12-0340-1*} {*AI12-0384-2*} The packages Strings.Text_Buffers, Strings.Text_Buffers.Unbounded, and Strings.Text_Buffers.Bounded are new.

A.5 The Numerics Packages

1 The library package Numerics is the parent of several child units that provide facilities for mathematical computation. One child, the generic package Generic_Elementary_Functions, is defined in A.5.1, together with nongeneric equivalents; two others, the package Float_Random and the generic package Discrete_Random, are defined in A.5.2. Additional (optional) children are defined in Annex G, "Numerics".

Static Semantics

2/1 *This paragraph was deleted.*

3/5
```
{AI95-00388-01} {AI12-0414-1} package Ada.Numerics
   with Pure is
   Argument_Error : exception;
   Pi : constant :=
          3.14159_26535_89793_23846_26433_83279_50288_41971_69399_37511;
   π  : constant := Pi;
   e  : constant :=
          2.71828_18284_59045_23536_02874_71352_66249_77572_47093_69996;
end Ada.Numerics;
```

4 The Argument_Error exception is raised by a subprogram in a child unit of Numerics to signal that one or more of the actual subprogram parameters are outside the domain of the corresponding mathematical function.

Implementation Permissions

5 The implementation may specify the values of Pi and e to a larger number of significant digits.

5.a **Reason:** 51 digits seem more than adequate for all present computers; converted to binary, the values given above are accurate to more than 160 bits. Nevertheless, the permission allows implementations to accommodate unforeseen hardware advances.

Extensions to Ada 83

5.b Numerics and its children were not predefined in Ada 83.

Extensions to Ada 95

5.c/2 {*AI95-00388-01*} The alternative declaration of π is new.

A.5.1 Elementary Functions

1 Implementation-defined approximations to the mathematical functions known as the "elementary functions" are provided by the subprograms in Numerics.Generic_Elementary_Functions. Nongeneric equivalents of this generic package for each of the predefined floating point types are also provided as children of Numerics.

1.a **Implementation defined:** The accuracy actually achieved by the elementary functions.

Static Semantics

2 The generic library package Numerics.Generic_Elementary_Functions has the following declaration:

```
{AI12-0241-1} generic                                                                    3/5
   type Float_Type is digits <>;

package Ada.Numerics.Generic_Elementary_Functions
   with Pure, Nonblocking is

   function Sqrt    (X           : Float_Type'Base) return Float_Type'Base;             4
   function Log     (X           : Float_Type'Base) return Float_Type'Base;
   function Log     (X, Base     : Float_Type'Base) return Float_Type'Base;
   function Exp     (X           : Float_Type'Base) return Float_Type'Base;
   function "**"    (Left, Right : Float_Type'Base) return Float_Type'Base;

   function Sin     (X           : Float_Type'Base) return Float_Type'Base;             5
   function Sin     (X, Cycle    : Float_Type'Base) return Float_Type'Base;
   function Cos     (X           : Float_Type'Base) return Float_Type'Base;
   function Cos     (X, Cycle    : Float_Type'Base) return Float_Type'Base;
   function Tan     (X           : Float_Type'Base) return Float_Type'Base;
   function Tan     (X, Cycle    : Float_Type'Base) return Float_Type'Base;
   function Cot     (X           : Float_Type'Base) return Float_Type'Base;
   function Cot     (X, Cycle    : Float_Type'Base) return Float_Type'Base;

   function Arcsin  (X           : Float_Type'Base) return Float_Type'Base;             6
   function Arcsin  (X, Cycle    : Float_Type'Base) return Float_Type'Base;
   function Arccos  (X           : Float_Type'Base) return Float_Type'Base;
   function Arccos  (X, Cycle    : Float_Type'Base) return Float_Type'Base;
   function Arctan  (Y           : Float_Type'Base;
                     X           : Float_Type'Base := 1.0)
                                                    return Float_Type'Base;
   function Arctan  (Y           : Float_Type'Base;
                     X           : Float_Type'Base := 1.0;
                     Cycle       : Float_Type'Base) return Float_Type'Base;
   function Arccot  (X           : Float_Type'Base;
                     Y           : Float_Type'Base := 1.0)
                                                    return Float_Type'Base;
   function Arccot  (X           : Float_Type'Base;
                     Y           : Float_Type'Base := 1.0;
                     Cycle       : Float_Type'Base) return Float_Type'Base;

   function Sinh    (X           : Float_Type'Base) return Float_Type'Base;             7
   function Cosh    (X           : Float_Type'Base) return Float_Type'Base;
   function Tanh    (X           : Float_Type'Base) return Float_Type'Base;
   function Coth    (X           : Float_Type'Base) return Float_Type'Base;
   function Arcsinh (X           : Float_Type'Base) return Float_Type'Base;
   function Arccosh (X           : Float_Type'Base) return Float_Type'Base;
   function Arctanh (X           : Float_Type'Base) return Float_Type'Base;
   function Arccoth (X           : Float_Type'Base) return Float_Type'Base;

end Ada.Numerics.Generic_Elementary_Functions;                                           8
```

{*8652/0020*} {*AI95-00126-01*} The library package Numerics.Elementary_Functions is declared pure and defines the same subprograms as Numerics.Generic_Elementary_Functions, except that the predefined type Float is systematically substituted for Float_Type'Base throughout. Nongeneric equivalents of Numerics.Generic_Elementary_Functions for each of the other predefined floating point types are defined similarly, with the names Numerics.Short_Elementary_Functions, Numerics.Long_Elementary_-Functions, etc. 9/1

Reason: The nongeneric equivalents are provided to allow the programmer to construct simple mathematical applications without being required to understand and use generics. 9.a

The functions have their usual mathematical meanings. When the Base parameter is specified, the Log function computes the logarithm to the given base; otherwise, it computes the natural logarithm. When the Cycle parameter is specified, the parameter X of the forward trigonometric functions (Sin, Cos, Tan, and Cot) and the results of the inverse trigonometric functions (Arcsin, Arccos, Arctan, and Arccot) are measured in units such that a full cycle of revolution has the given value; otherwise, they are measured in radians. 10

11 The computed results of the mathematically multivalued functions are rendered single-valued by the following conventions, which are meant to imply the principal branch:

12 • The results of the Sqrt and Arccosh functions and that of the exponentiation operator are nonnegative.

13 • The result of the Arcsin function is in the quadrant containing the point (1.0, *x*), where *x* is the value of the parameter X. This quadrant is I or IV; thus, the range of the Arcsin function is approximately –π/2.0 to π/2.0 (–Cycle/4.0 to Cycle/4.0, if the parameter Cycle is specified).

14 • The result of the Arccos function is in the quadrant containing the point (*x*, 1.0), where *x* is the value of the parameter X. This quadrant is I or II; thus, the Arccos function ranges from 0.0 to approximately π (Cycle/2.0, if the parameter Cycle is specified).

15 • The results of the Arctan and Arccot functions are in the quadrant containing the point (*x*, *y*), where *x* and *y* are the values of the parameters X and Y, respectively. This may be any quadrant (I through IV) when the parameter X (resp., Y) of Arctan (resp., Arccot) is specified, but it is restricted to quadrants I and IV (resp., I and II) when that parameter is omitted. Thus, the range when that parameter is specified is approximately –π to π (–Cycle/2.0 to Cycle/2.0, if the parameter Cycle is specified); when omitted, the range of Arctan (resp., Arccot) is that of Arcsin (resp., Arccos), as given above. When the point (*x*, *y*) lies on the negative x-axis, the result approximates

16 – π (resp., –π) when the sign of the parameter Y is positive (resp., negative), if Float_Type'Signed_Zeros is True;

17 – π, if Float_Type'Signed_Zeros is False.

18 (In the case of the inverse trigonometric functions, in which a result lying on or near one of the axes may not be exactly representable, the approximation inherent in computing the result may place it in an adjacent quadrant, close to but on the wrong side of the axis.)

Dynamic Semantics

19 The exception Numerics.Argument_Error is raised, signaling a parameter value outside the domain of the corresponding mathematical function, in the following cases:

20 • by any forward or inverse trigonometric function with specified cycle, when the value of the parameter Cycle is zero or negative;

21 • by the Log function with specified base, when the value of the parameter Base is zero, one, or negative;

22 • by the Sqrt and Log functions, when the value of the parameter X is negative;

23 • by the exponentiation operator, when the value of the left operand is negative or when both operands have the value zero;

24 • by the Arcsin, Arccos, and Arctanh functions, when the absolute value of the parameter X exceeds one;

25 • by the Arctan and Arccot functions, when the parameters X and Y both have the value zero;

26 • by the Arccosh function, when the value of the parameter X is less than one; and

27 • by the Arccoth function, when the absolute value of the parameter X is less than one.

28 The exception Constraint_Error is raised, signaling a pole of the mathematical function (analogous to dividing by zero), in the following cases, provided that Float_Type'Machine_Overflows is True:

29 • by the Log, Cot, and Coth functions, when the value of the parameter X is zero;

30 • by the exponentiation operator, when the value of the left operand is zero and the value of the exponent is negative;

- by the Tan function with specified cycle, when the value of the parameter X is an odd multiple of the quarter cycle; 31
- by the Cot function with specified cycle, when the value of the parameter X is zero or a multiple of the half cycle; and 32
- by the Arctanh and Arccoth functions, when the absolute value of the parameter X is one. 33

[Constraint_Error can also be raised when a finite result overflows (see G.2.4); this may occur for parameter values sufficiently *near* poles, and, in the case of some of the functions, for parameter values with sufficiently large magnitudes.] When Float_Type'Machine_Overflows is False, the result at poles is unspecified. 34

Reason: The purpose of raising Constraint_Error (rather than Numerics.Argument_Error) at the poles of a function, when Float_Type'Machine_Overflows is True, is to provide continuous behavior as the actual parameters of the function approach the pole and finally reach it. 34.a

Discussion: It is anticipated that an Ada binding to IEC 559:1989 will be developed in the future. As part of such a binding, the Machine_Overflows attribute of a conformant floating point type will be specified to yield False, which will permit both the predefined arithmetic operations and implementations of the elementary functions to deliver signed infinities (and set the overflow flag defined by the binding) instead of raising Constraint_Error in overflow situations, when traps are disabled. Similarly, it is appropriate for the elementary functions to deliver signed infinities (and set the zero-divide flag defined by the binding) instead of raising Constraint_Error at poles, when traps are disabled. Finally, such a binding should also specify the behavior of the elementary functions, when sensible, given parameters with infinite values. 34.b

{*AI12-0453-1*} When one parameter of a function with multiple parameters represents a pole and another is outside the function's domain, the latter takes precedence and Numerics.Argument_Error is raised. 35/5

Implementation Requirements

In the implementation of Numerics.Generic_Elementary_Functions, the range of intermediate values allowed during the calculation of a final result shall not be affected by any range constraint of the subtype Float_Type. 36

Implementation Note: Implementations of Numerics.Generic_Elementary_Functions written in Ada should therefore avoid declaring local variables of subtype Float_Type; the subtype Float_Type'Base should be used instead. 36.a

In the following cases, evaluation of an elementary function shall yield the *prescribed result*, provided that the preceding rules do not call for an exception to be raised: 37

- When the parameter X has the value zero, the Sqrt, Sin, Arcsin, Tan, Sinh, Arcsinh, Tanh, and Arctanh functions yield a result of zero, and the Exp, Cos, and Cosh functions yield a result of one. 38
- When the parameter X has the value one, the Sqrt function yields a result of one, and the Log, Arccos, and Arccosh functions yield a result of zero. 39
- When the parameter Y has the value zero and the parameter X has a positive value, the Arctan and Arccot functions yield a result of zero. 40
- The results of the Sin, Cos, Tan, and Cot functions with specified cycle are exact when the mathematical result is zero; those of the first two are also exact when the mathematical result is ± 1.0. 41
- Exponentiation by a zero exponent yields the value one. Exponentiation by a unit exponent yields the value of the left operand. Exponentiation of the value one yields the value one. Exponentiation of the value zero yields the value zero. 42

Other accuracy requirements for the elementary functions, which apply only in implementations conforming to the Numerics Annex, and then only in the “strict” mode defined there (see G.2), are given in G.2.4. 43

44 When Float_Type'Signed_Zeros is True, the sign of a zero result shall be as follows:

45 • A prescribed zero result delivered *at the origin* by one of the odd functions (Sin, Arcsin, Sinh, Arcsinh, Tan, Arctan or Arccot as a function of Y when X is fixed and positive, Tanh, and Arctanh) has the sign of the parameter X (Y, in the case of Arctan or Arccot).

46 • A prescribed zero result delivered by one of the odd functions *away from the origin*, or by some other elementary function, has an implementation-defined sign.

46.a **Implementation defined:** The sign of a zero result from some of the operators or functions in Numerics.Generic_Elementary_Functions, when Float_Type'Signed_Zeros is True.

47/5 • {*AI12-0453-1*} [A zero result that is not a prescribed result (that is, one that results from rounding or underflow) has the correct mathematical sign.]

47.a **Reason:** This is a consequence of the rules specified in IEC 559:1989 as they apply to underflow situations with traps disabled.

Implementation Permissions

48/5 {*AI12-0444-1*} The nongeneric equivalent packages can be actual instantiations of the generic package for the appropriate predefined type, though that is not required.

Wording Changes from Ada 83

48.a The semantics of Numerics.Generic_Elementary_Functions differs from Generic_Elementary_Functions as defined in ISO/IEC DIS 11430 (for Ada 83) in the following ways:

48.b • The generic package is a child unit of the package defining the Argument_Error exception.

48.c • DIS 11430 specified names for the nongeneric equivalents, if provided. Here, those nongeneric equivalents are required.

48.d • Implementations are not allowed to impose an optional restriction that the generic actual parameter associated with Float_Type be unconstrained. (In view of the ability to declare variables of subtype Float_Type'Base in implementations of Numerics.Generic_Elementary_Functions, this flexibility is no longer needed.)

48.e • The sign of a prescribed zero result at the origin of the odd functions is specified, when Float_Type'Signed_Zeros is True. This conforms with recommendations of Kahan and other numerical analysts.

48.f • The dependence of Arctan and Arccot on the sign of a parameter value of zero is tied to the value of Float_Type'Signed_Zeros.

48.g • Sqrt is prescribed to yield a result of one when its parameter has the value one. This guarantee makes it easier to achieve certain prescribed results of the complex elementary functions (see G.1.2, “Complex Elementary Functions”).

48.h • Conformance to accuracy requirements is conditional.

Wording Changes from Ada 95

48.i/2 {*8652/0020*} {*AI95-00126-01*} **Corrigendum:** Explicitly stated that the nongeneric equivalents of Generic_Elementary_Functions are pure.

A.5.2 Random Number Generation

1 [Facilities for the generation of pseudo-random floating point numbers are provided in the package Numerics.Float_Random; the generic package Numerics.Discrete_Random provides similar facilities for the generation of pseudo-random integers and pseudo-random values of enumeration types. For brevity, pseudo-random values of any of these types are called *random numbers*.

2 Some of the facilities provided are basic to all applications of random numbers. These include a limited private type each of whose objects serves as the generator of a (possibly distinct) sequence of random numbers; a function to obtain the “next” random number from a given sequence of random numbers (that

is, from its generator); and subprograms to initialize or reinitialize a given generator to a time-dependent state or a state denoted by a single integer.

Other facilities are provided specifically for advanced applications. These include subprograms to save and restore the state of a given generator; a private type whose objects can be used to hold the saved state of a generator; and subprograms to obtain a string representation of a given generator state, or, given such a string representation, the corresponding state.] 3

Discussion: These facilities support a variety of requirements ranging from repeatable sequences (for debugging) to unique sequences in each execution of a program. 3.a

Static Semantics

The library package Numerics.Float_Random has the following declaration: 4

```
{AI12-0302-1} package Ada.Numerics.Float_Random                                    5/5
   with Global => in out synchronized is

   -- Basic facilities                                                            6

   type Generator is limited private;                                            7

{AI12-0302-1}    subtype Uniformly_Distributed is Float range 0.0 .. 1.0;        8/5
   function Random (Gen : Generator) return Uniformly_Distributed
      with Global => overriding in out Gen;

{AI12-0302-1}    procedure Reset (Gen       : in Generator;                      9/5
                                  Initiator : in Integer)
      with Global => overriding in out Gen;
   procedure Reset (Gen       : in Generator)
      with Global => overriding in out Gen;

   -- Advanced facilities                                                        10

   type State is private;                                                        11

{AI12-0302-1}    procedure Save  (Gen        : in  Generator;                   12/5
                                  To_State   : out State);
   procedure Reset (Gen        : in  Generator;
                    From_State : in  State)
      with Global => overriding in out Gen;

   Max_Image_Width : constant := implementation-defined integer value;           13

   function Image (Of_State    : State)  return String;                          14
   function Value (Coded_State : String) return State;

private                                                                          15
   ... -- not specified by the language
end Ada.Numerics.Float_Random;
```

{*AI95-00360-01*} The type Generator needs finalization (see 7.6). 15.1/2

The generic library package Numerics.Discrete_Random has the following declaration: 16

```
{AI12-0302-1} generic                                                            17/5
   type Result_Subtype is (<>);
package Ada.Numerics.Discrete_Random
   with Global => in out synchronized is

   -- Basic facilities                                                           18

   type Generator is limited private;                                            19

{AI12-0302-1}    function Random (Gen : Generator) return Result_Subtype        20/5
      with Global => overriding in out Gen;

{AI12-0144-1} {AI12-0302-1}    function Random (Gen   : Generator;               20.1/5
                               First : Result_Subtype;
                               Last  : Result_Subtype) return Result_Subtype
      with Post => Random'Result in First .. Last,
           Global => overriding in out Gen;
```

21/5 {*AI12-0302-1*}
```
   procedure Reset (Gen       : in Generator;
                    Initiator : in Integer)
      with Global => overriding in out Gen;
   procedure Reset (Gen       : in Generator)
      with Global => overriding in out Gen;
```

22
```
   -- Advanced facilities
```

23
```
   type State is private;
```

24/5 {*AI12-0302-1*}
```
   procedure Save  (Gen        : in  Generator;
                    To_State   : out State);
   procedure Reset (Gen        : in  Generator;
                    From_State : in  State)
      with Global => overriding in out Gen;
```

25
```
   Max_Image_Width : constant := implementation-defined integer value;
```

26
```
   function Image (Of_State    : State)  return String;
   function Value (Coded_State : String) return State;
```

27
```
private
   ... -- not specified by the language
end Ada.Numerics.Discrete_Random;
```

27.a **Implementation defined:** The value of Numerics.Float_Random.Max_Image_Width.

27.b **Implementation defined:** The value of Numerics.Discrete_Random.Max_Image_Width.

27.c/1 **Implementation Note:** {*8652/0097*} {*AI95-00115-01*} The following is a possible implementation of the private part of Numerics.Float_Random (assuming the presence of "**with** Ada.Finalization;" as a context clause):

27.d
```
type State is ...;
type Access_State is access State;
type Generator is new Finalization.Limited_Controlled with
   record
      S : Access_State := new State'(...);
   end record;
procedure Finalize (G : in out Generator);
```

27.d.1/2 {*8652/0097*} {*AI95-00115-01*} {*AI95-00344-01*} Numerics.Discrete_Random.Generator also can be implemented this way.

27.e Clearly some level of indirection is required in the implementation of a Generator, since the parameter mode is **in** for all operations on a Generator. For this reason, Numerics.Float_Random and Numerics.Discrete_Random cannot be declared pure.

27.1/2 {*AI95-00360-01*} The type Generator needs finalization (see 7.6) in every instantiation of Numerics.Discrete_Random.

28 An object of the limited private type Generator is associated with a sequence of random numbers. Each generator has a hidden (internal) state, which the operations on generators use to determine the position in the associated sequence. All generators are implicitly initialized to an unspecified state that does not vary from one program execution to another; they may also be explicitly initialized, or reinitialized, to a time-dependent state, to a previously saved state, or to a state uniquely denoted by an integer value.

28.a **Discussion:** The repeatability provided by the implicit initialization may be exploited for testing or debugging purposes.

29/5 {*AI05-0280-1*} {*AI12-0445-1*} An object of the private type State can be used to hold the internal state of a generator. Such objects are only necessary if the application is designed to save and restore generator states or to examine or manufacture them. The implicit initial value of type State corresponds to the implicit initial value of all generators.

29.a/3 **Discussion:** {*AI05-0280-1*} All generators are implicitly initialized to the same unchanging value, and using Reset on a default initialized object of type State will produce a generator with that same value.

30 The operations on generators affect the state and therefore the future values of the associated sequence. The semantics of the operations on generators and states are defined below.

31
```
function Random (Gen : Generator) return Uniformly_Distributed;
function Random (Gen : Generator) return Result_Subtype;
```

{*AI12-0144-1*} Obtains the "next" random number from the given generator, relative to its current state, according to an implementation-defined algorithm. 32/5

This paragraph was deleted. 32.a/2

Discussion: The algorithm is the subject of a Documentation Requirement, so we don't separately summarize this implementation-defined item. 32.a.1/2

Reason: The requirement for a level of indirection in accessing the internal state of a generator arises from the desire to make Random a function, rather than a procedure. 32.b

```
{AI12-0144-1} function Random (Gen   : Generator;                                    32.1/5
                               First : Result_Subtype;
                               Last  : Result_Subtype) return Result_Subtype
   with Post => Random'Result in First .. Last;
```

{*AI12-0144-1*} Obtains the "next" random number from the given generator, relative to its current state, according to an implementation-defined algorithm. If the range First .. Last is a null range, Constraint_Error is raised. 32.2/5

```
procedure Reset (Gen       : in Generator;                                           33
                 Initiator : in Integer);
procedure Reset (Gen       : in Generator);
```

Sets the state of the specified generator to one that is an unspecified function of the value of the parameter Initiator (or to a time-dependent state, if only a generator parameter is specified). The latter form of the procedure is known as the *time-dependent Reset procedure*. 34

Implementation Note: The time-dependent Reset procedure can be implemented by mapping the current time and date as determined by the system clock into a state, but other implementations are possible. For example, a white-noise generator or a radioactive source can be used to generate time-dependent states. 34.a

```
procedure Save  (Gen        : in  Generator;                                         35
                 To_State   : out State);
procedure Reset (Gen        : in  Generator;
                 From_State : in  State);
```

Save obtains the current state of a generator. Reset gives a generator the specified state. A generator that is reset to a state previously obtained by invoking Save is restored to the state it had when Save was invoked. 36

```
function Image (Of_State    : State)  return String;                                 37
function Value (Coded_State : String) return State;
```

Image provides a representation of a state coded (in an implementation-defined way) as a string whose length is bounded by the value of Max_Image_Width. Value is the inverse of Image: Value(Image(S)) = S for each state S that can be obtained from a generator by invoking Save. 38

Implementation defined: The string representation of a random number generator's state. 38.a

Dynamic Semantics

Instantiation of Numerics.Discrete_Random with a subtype having a null range raises Constraint_Error. 39

This paragraph was deleted.{*8652/0050*} {*AI95-00089*} 40/1

Bounded (Run-Time) Errors

{*8652/0050*} {*AI95-00089*} {*AI12-0445-1*} It is a bounded error to invoke Value with a string that is not the image of any generator state. If the error is detected, Constraint_Error or Program_Error is raised. Otherwise, a call to Reset with the resulting state will produce a generator such that calls to Random with this generator will produce a sequence of values of the appropriate subtype, but which are not necessarily random in character. That is, the sequence of values do not necessarily fulfill the implementation requirements of this subclause. 40.1/5

Implementation Requirements

40.2/5 {*AI12-0144-1*} Each call of a Random function has a *result range*; this is the range First .. Last for the version of Random with First and Last parameters and the range of the result subtype of the function otherwise.

41/5 {*AI12-0144-1*} A sufficiently long sequence of random numbers obtained by consecutive calls to Random that have the same generator and result range is approximately uniformly distributed over the result range.

41.a/5 **Discussion:** {*AI12-0005-1*} {*AI12-0144-1*} In this rule, "consecutive" means at least that there are no intervening explicit calls involving the same generator. This restricts the rule to only applying to cases where just the Random function changes the generator. We don't mean to impose a requirement if there are intervening calls to Reset, to Random with the same generator but a different result range, or any other case that would affect the sequence of values returned. Operations that use the resulting random values (for instance, to store them somewhere) are not considered in determining if calls are consecutive.

42/5 {*AI12-0144-1*} A Random function in an instantiation of Numerics.Discrete_Random is guaranteed to yield each value in its result range in a finite number of calls, provided that the number of such values does not exceed 2^{15}.

43 Other performance requirements for the random number generator, which apply only in implementations conforming to the Numerics Annex, and then only in the "strict" mode defined there (see G.2), are given in G.2.5.

Documentation Requirements

44 No one algorithm for random number generation is best for all applications. To enable the user to determine the suitability of the random number generators for the intended application, the implementation shall describe the algorithm used and shall give its period, if known exactly, or a lower bound on the period, if the exact period is unknown. Periods that are so long that the periodicity is unobservable in practice can be described in such terms, without giving a numerical bound.

44.a/2 **Documentation Requirement:** The algorithm used for random number generation, including a description of its period.

45 The implementation also shall document the minimum time interval between calls to the time-dependent Reset procedure that are guaranteed to initiate different sequences, and it shall document the nature of the strings that Value will accept without raising Constraint_Error.

45.a/2 *This paragraph was deleted.*

45.b/2 **Documentation Requirement:** The minimum time interval between calls to the time-dependent Reset procedure that is guaranteed to initiate different random number sequences.

Implementation Advice

46 Any storage associated with an object of type Generator should be reclaimed on exit from the scope of the object.

46.a.1/2 **Implementation Advice:** Any storage associated with an object of type Generator of the random number packages should be reclaimed on exit from the scope of the object.

46.a **Ramification:** A level of indirection is implicit in the semantics of the operations, given that they all take parameters of mode **in**. This implies that the full type of Generator probably should be a controlled type, with appropriate finalization to reclaim any heap-allocated storage.

47 If the generator period is sufficiently long in relation to the number of distinct initiator values, then each possible value of Initiator passed to Reset should initiate a sequence of random numbers that does not, in a practical sense, overlap the sequence initiated by any other value. If this is not possible, then the mapping between initiator values and generator states should be a rapidly varying function of the initiator value.

Implementation Advice: Each value of Initiator passed to Reset for the random number packages should initiate a distinct sequence of random numbers, or, if that is not possible, be at least a rapidly varying function of the initiator value. 47.a/2

NOTE 1 If two or more tasks are to share the same generator, then the tasks have to synchronize their access to the generator as for any shared variable (see 9.10). 48

NOTE 2 Within a given implementation, a repeatable random number sequence can be obtained by relying on the implicit initialization of generators or by explicitly initializing a generator with a repeatable initiator value. Different sequences of random numbers can be obtained from a given generator in different program executions by explicitly initializing the generator to a time-dependent state. 49

NOTE 3 {*AI12-0442-1*} {*AI12-0447-1*} A given implementation of the Random function in Numerics.Float_Random is not guaranteed to be capable of delivering the values 0.0 or 1.0. Applications will be more portable if they assume that these values, or values sufficiently close to them to behave indistinguishably from them, can occur. If a sequence of random integers from some range is necessary, the application should use one of the Random functions in an appropriate instantiation of Numerics.Discrete_Random, rather than transforming the result of the Random function in Numerics.Float_Random. 50/5

This paragraph was deleted.{*AI12-0442-1*} 51/5

Reason: {*AI12-0442-1*} One might think that a simple transformation of the result of the floating point Random function such as `Integer(Float(M) * Random(G)) mod M` would give a uniform distribution. But this is only true if the period of the underlying generator is a multiple of `M`. (This usually requires that `M` be a power of two.) In other cases, the **mod** operation maps slightly more random values to some result values than others. It is easy to see this: consider a 4-bit random integer (with a range of 0 .. 15). If one **mod**s this by 6 to get a value in 0 .. 5 (to which one would add 1 to get the value of a die roll), 3 values would be mapped to each value 0 .. 3, but only 2 values would be mapped to 4 and 5. Even when the input is uniformly distributed, the output clearly is not. A similar effect occurs regardless of the number of bits in the random integer. Since it takes care to get this right, users should use the provided functions (which presumably do this correctly – AI12-0144-1 contains a correct algorithm) and resist the urge to "roll-their-own". 51.a/5

NOTE 4 {*AI12-0442-1*} Exponentially distributed (floating point) random numbers with mean and standard deviation 1.0 can be obtained by the transformation 52/5

{*AI95-00434-01*} `-Log(Random(G) + Float'Model_Small)` 53/2

where Log comes from Numerics.Elementary_Functions (see A.5.1); in this expression, the addition of Float'Model_Small avoids the exception that would be raised were Log to be given the value zero, without affecting the result (in most implementations) when Random returns a nonzero value. 54

Examples

Example of a program that plays a simulated dice game: 55

```
with Ada.Numerics.Discrete_Random;                                    56
procedure Dice_Game is
   subtype Die is Integer range 1 .. 6;
   subtype Dice is Integer range 2*Die'First .. 2*Die'Last;
   package Random_Die is new Ada.Numerics.Discrete_Random (Die);
   use Random_Die;
   G : Generator;
   D : Dice;
begin
   Reset (G);  -- Start the generator in a unique state in each run
   loop
      -- Roll a pair of dice; sum and process the results
      D := Random(G) + Random(G);
      ...
   end loop;
end Dice_Game;
```

57 *Example of a program that simulates coin tosses:*

58
```
with Ada.Numerics.Discrete_Random;
procedure Flip_A_Coin is
   type Coin is (Heads, Tails);
   package Random_Coin is new Ada.Numerics.Discrete_Random (Coin);
   use Random_Coin;
   G : Generator;
begin
   Reset (G);  -- Start the generator in a unique state in each run
   loop
      -- Toss a coin and process the result
      case Random(G) is
          when Heads =>
             ...
          when Tails =>
             ...
      end case;
   ...
   end loop;
end Flip_A_Coin;
```

59 *Example of a parallel simulation of a physical system, with a separate generator of event probabilities in each task:*

60
```
with Ada.Numerics.Float_Random;
procedure Parallel_Simulation is
   use Ada.Numerics.Float_Random;
   task type Worker is
      entry Initialize_Generator (Initiator : in Integer);
      ...
   end Worker;
   W : array (1 .. 10) of Worker;
   task body Worker is
      G : Generator;
      Probability_Of_Event : Uniformly_Distributed;
   begin
      accept Initialize_Generator (Initiator : in Integer) do
         Reset (G, Initiator);
      end Initialize_Generator;
      loop
         ...
         Probability_Of_Event := Random(G);
         ...
      end loop;
   end Worker;
begin
   -- Initialize the generators in the Worker tasks to different states
   for I in W'Range loop
      W(I).Initialize_Generator (I);
   end loop;
   ... -- Wait for the Worker tasks to terminate
end Parallel_Simulation;
```

61/5 {*AI12-0452-1*} Although each Worker task initializes its generator to a different state, those states will be the same in every execution of the program. The generator states can be initialized uniquely in each program execution by instantiating Ada.Numerics.Discrete_Random for the type Integer in the main procedure, resetting the generator obtained from that instance to a time-dependent state, and then using random integers obtained from that generator to initialize the generators in each Worker task.

Incompatibilities With Ada 95

62.a/2 {*AI95-00360-01*} **Amendment Correction:** Type Generator in Numerics.Float_Random and in an instance of Numerics.Discrete_Random is defined to need finalization. If the restriction No_Nested_Finalization (see D.7) applies to the partition, and Generator does not have a controlled part, it will not be allowed in local objects in Ada 2005

whereas it would be allowed in original Ada 95. Such code is not portable, as another Ada compiler may have a controlled part in Generator, and thus would be illegal.

Wording Changes from Ada 95

{*8652/0050*} {*AI95-00089-01*} {*AI05-0005-1*} **Corrigendum:** Made the passing of an incorrect Image of a generator a bounded error, as it might not be practical to check for problems (if a generator consists of several related values). 62.b/3

Wording Changes from Ada 2005

{*AI05-0280-1*} **Correction:** Specified the implicit initial value for (sub)type State. This was unspecified in Ada 95 and Ada 2005, so a program depending on some other initial value is very unlikely and certainly was not portable. An implementation can use default expressions, aspect Default_Value, or aspect Default_Component_Value to keep the representation of the type unchanged while meeting this new requirement. 62.c/3

Extensions to Ada 2012

{*AI12-0144-1*} The function Random with First and Last parameters is new. 62.d/5

A.5.3 Attributes of Floating Point Types

Static Semantics

The following *representation-oriented attributes* are defined for every subtype S of a floating point type *T*. 1

S'Machine_Radix 2

Yields the radix of the hardware representation of the type *T*. The value of this attribute is of the type *universal_integer*.

The values of other representation-oriented attributes of a floating point subtype, and of the "primitive function" attributes of a floating point subtype described later, are defined in terms of a particular representation of nonzero values called the *canonical form*. The canonical form (for the type *T*) is the form 3

$$\pm\, mantissa \cdot T\text{'Machine\_Radix}^{exponent}$$

where

- *mantissa* is a fraction in the number base *T*'Machine_Radix, the first digit of which is nonzero, and 4
- *exponent* is an integer. 5

S'Machine_Mantissa 6

Yields the largest value of *p* such that every value expressible in the canonical form (for the type *T*), having a *p*-digit *mantissa* and an *exponent* between *T*'Machine_Emin and *T*'Machine_Emax, is a machine number (see 3.5.7) of the type *T*. This attribute yields a value of the type *universal_integer*.

Ramification: Values of a type held in an extended register are, in general, not machine numbers of the type, since they cannot be expressed in the canonical form with a sufficiently short *mantissa*. 6.a

S'Machine_Emin 7

Yields the smallest (most negative) value of *exponent* such that every value expressible in the canonical form (for the type *T*), having a *mantissa* of *T*'Machine_Mantissa digits, is a machine number (see 3.5.7) of the type *T*. This attribute yields a value of the type *universal_integer*.

S'Machine_Emax 8

Yields the largest (most positive) value of *exponent* such that every value expressible in the canonical form (for the type *T*), having a *mantissa* of *T*'Machine_Mantissa digits, is a machine number (see 3.5.7) of the type *T*. This attribute yields a value of the type *universal_integer*.

8.a **Ramification:** Note that the above definitions do not determine unique values for the representation-oriented attributes of floating point types. The implementation may choose any set of values that collectively satisfies the definitions.

9 S'Denorm Yields the value True if every value expressible in the form

$$\pm \textit{mantissa} \cdot \textit{T}\text{'Machine\_Radix}^{\textit{T}\text{'Machine\_Emin}}$$

where *mantissa* is a nonzero *T*'Machine_Mantissa-digit fraction in the number base *T*'Machine_Radix, the first digit of which is zero, is a machine number (see 3.5.7) of the type *T*; yields the value False otherwise. The value of this attribute is of the predefined type Boolean.

10 The values described by the formula in the definition of S'Denorm are called *denormalized numbers*. A nonzero machine number that is not a denormalized number is a *normalized number*. A normalized number *x* of a given type *T* is said to be *represented in canonical form* when it is expressed in the canonical form (for the type *T*) with a *mantissa* having *T*'Machine_Mantissa digits; the resulting form is the *canonical-form representation* of *x*.

10.a **Discussion:** The intent is that S'Denorm be True when such denormalized numbers exist and are generated in the circumstances defined by IEC 559:1989, though the latter requirement is not formalized here.

11 S'Machine_Rounds
Yields the value True if rounding is performed on inexact results of every predefined operation that yields a result of the type *T*; yields the value False otherwise. The value of this attribute is of the predefined type Boolean.

11.a **Discussion:** It is difficult to be more precise about what it means to round the result of a predefined operation. If the implementation does not use extended registers, so that every arithmetic result is necessarily a machine number, then rounding seems to imply two things:

11.b - S'Model_Mantissa = S'Machine_Mantissa, so that operand preperturbation never occurs;

11.c - when the exact mathematical result is not a machine number, the result of a predefined operation must be the nearer of the two adjacent machine numbers.

11.d Technically, this attribute should yield False when extended registers are used, since a few computed results will cross over the half-way point as a result of double rounding, if and when a value held in an extended register has to be reduced in precision to that of the machine numbers. It does not seem desirable to preclude the use of extended registers when S'Machine_Rounds could otherwise be True.

12 S'Machine_Overflows
Yields the value True if overflow and divide-by-zero are detected and reported by raising Constraint_Error for every predefined operation that yields a result of the type *T*; yields the value False otherwise. The value of this attribute is of the predefined type Boolean.

13 S'Signed_Zeros
Yields the value True if the hardware representation for the type *T* has the capability of representing both positively and negatively signed zeros, these being generated and used by the predefined operations of the type *T* as specified in IEC 559:1989; yields the value False otherwise. The value of this attribute is of the predefined type Boolean.

14 For every value *x* of a floating point type *T*, the *normalized exponent* of *x* is defined as follows:

15 - the normalized exponent of zero is (by convention) zero;

16 - for nonzero *x*, the normalized exponent of *x* is the unique integer *k* such that $\textit{T}\text{'Machine\_Radix}^{k-1} \le |x| < \textit{T}\text{'Machine\_Radix}^{k}$.

16.a **Ramification:** The normalized exponent of a normalized number *x* is the value of *exponent* in the canonical-form representation of *x*.

16.b The normalized exponent of a denormalized number is less than the value of *T*'Machine_Emin.

17 The following *primitive function attributes* are defined for any subtype S of a floating point type *T*.

18 S'Exponent S'Exponent denotes a function with the following specification:

19
```
function S'Exponent (X : T)
  return universal_integer
```

The function yields the normalized exponent of *X*. 20

S'Fraction S'Fraction denotes a function with the following specification: 21

```
function S'Fraction (X : T)
  return T
```
22

The function yields the value $X \cdot T\text{'Machine\_Radix}^{-k}$, where *k* is the normalized exponent of *X*. A zero result[, which can only occur when *X* is zero,] has the sign of *X*. 23

Discussion: Informally, when *X* is a normalized number, the result is the value obtained by replacing the *exponent* by zero in the canonical-form representation of *X*. 23.a

Ramification: Except when *X* is zero, the magnitude of the result is greater than or equal to the reciprocal of *T*'Machine_Radix and less than one; consequently, the result is always a normalized number, even when *X* is a denormalized number. 23.b

Implementation Note: When *X* is a denormalized number, the result is the value obtained by replacing the *exponent* by zero in the canonical-form representation of the result of scaling *X* up sufficiently to normalize it. 23.c

S'Compose S'Compose denotes a function with the following specification: 24

```
function S'Compose (Fraction : T;
                    Exponent : universal_integer)
  return T
```
25

Let *v* be the value $Fraction \cdot T\text{'Machine\_Radix}^{Exponent-k}$, where *k* is the normalized exponent of *Fraction*. If *v* is a machine number of the type *T*, or if $|v| \geq T\text{'Model\_Small}$, the function yields *v*; otherwise, it yields either one of the machine numbers of the type *T* adjacent to *v*. Constraint_Error is optionally raised if *v* is outside the base range of S. A zero result has the sign of *Fraction* when S'Signed_Zeros is True. 26

Discussion: Informally, when *Fraction* and *v* are both normalized numbers, the result is the value obtained by replacing the *exponent* by *Exponent* in the canonical-form representation of *Fraction*. 26.a

Ramification: If *Exponent* is less than *T*'Machine_Emin and *Fraction* is nonzero, the result is either zero, *T*'Model_Small, or (if *T*'Denorm is True) a denormalized number. 26.b

S'Scaling S'Scaling denotes a function with the following specification: 27

```
function S'Scaling (X : T;
                    Adjustment : universal_integer)
  return T
```
28

Let *v* be the value $X \cdot T\text{'Machine\_Radix}^{Adjustment}$. If *v* is a machine number of the type *T*, or if $|v| \geq T\text{'Model\_Small}$, the function yields *v*; otherwise, it yields either one of the machine numbers of the type *T* adjacent to *v*. Constraint_Error is optionally raised if *v* is outside the base range of S. A zero result has the sign of *X* when S'Signed_Zeros is True. 29

Discussion: Informally, when *X* and *v* are both normalized numbers, the result is the value obtained by increasing the *exponent* by *Adjustment* in the canonical-form representation of *X*. 29.a

Ramification: {*AI12-0453-1*} If *Adjustment* is sufficiently small (that is, sufficiently negative), the result is either zero, *T*'Model_Small, or (if *T*'Denorm is True) a denormalized number. 29.b/5

S'Floor S'Floor denotes a function with the following specification: 30

```
function S'Floor (X : T)
  return T
```
31

{*AI12-0453-1*} The function yields the value $\lfloor X \rfloor$, that is, the largest (most positive) integral value less than or equal to *X*. When *X* is zero, the result has the sign of *X*; a zero result otherwise has a positive sign. 32/5

S'Ceiling S'Ceiling denotes a function with the following specification: 33

```
function S'Ceiling (X : T)
  return T
```
34

{*AI12-0453-1*} The function yields the value $\lceil X \rceil$, that is, the smallest (most negative) integral value greater than or equal to *X*. When *X* is zero, the result has the sign of *X*; a zero result otherwise has a negative sign when S'Signed_Zeros is True. 35/5

36 S'Rounding S'Rounding denotes a function with the following specification:

37
```
function S'Rounding (X : T)
  return T
```

38 The function yields the integral value nearest to *X*, rounding away from zero if *X* lies exactly halfway between two integers. A zero result has the sign of *X* when S'Signed_Zeros is True.

39 S'Unbiased_Rounding
S'Unbiased_Rounding denotes a function with the following specification:

40
```
function S'Unbiased_Rounding (X : T)
  return T
```

41 The function yields the integral value nearest to *X*, rounding toward the even integer if *X* lies exactly halfway between two integers. A zero result has the sign of *X* when S'Signed_Zeros is True.

41.1/2 S'Machine_Rounding
{*AI95-00267-01*} S'Machine_Rounding denotes a function with the following specification:

41.2/2
```
function S'Machine_Rounding (X : T)
  return T
```

41.3/2 The function yields the integral value nearest to *X*. If *X* lies exactly halfway between two integers, one of those integers is returned, but which of them is returned is unspecified. A zero result has the sign of *X* when S'Signed_Zeros is True. This function provides access to the rounding behavior which is most efficient on the target processor.

41.a.1/2 **Discussion:** We leave the rounding unspecified, so that users cannot depend on a particular rounding. This attribute is intended for use in cases where the particular rounding chosen is irrelevant. If there is a need to know which way values halfway between two integers are rounded, one of the other rounding attributes should be used.

42 S'Truncation
S'Truncation denotes a function with the following specification:

43
```
function S'Truncation (X : T)
  return T
```

44 The function yields the value $\lceil X \rceil$ when *X* is negative, and $\lfloor X \rfloor$ otherwise. A zero result has the sign of *X* when S'Signed_Zeros is True.

45 S'Remainder
S'Remainder denotes a function with the following specification:

46
```
function S'Remainder (X, Y : T)
  return T
```

47 For nonzero *Y*, let *v* be the value $X - n \cdot Y$, where *n* is the integer nearest to the exact value of *X*/*Y*; if $|n - X/Y| = 1/2$, then *n* is chosen to be even. If *v* is a machine number of the type *T*, the function yields *v*; otherwise, it yields zero. Constraint_Error is raised if *Y* is zero. A zero result has the sign of *X* when S'Signed_Zeros is True.

47.a **Ramification:** The magnitude of the result is less than or equal to one-half the magnitude of *Y*.

47.b **Discussion:** Given machine numbers *X* and *Y* of the type *T*, *v* is necessarily a machine number of the type *T*, except when *Y* is in the neighborhood of zero, *X* is sufficiently close to a multiple of *Y*, and *T*'Denorm is False.

48 S'Adjacent S'Adjacent denotes a function with the following specification:

49
```
function S'Adjacent (X, Towards : T)
  return T
```

50 If *Towards* = *X*, the function yields *X*; otherwise, it yields the machine number of the type *T* adjacent to *X* in the direction of *Towards*, if that machine number exists. If the result would be outside the base range of S, Constraint_Error is raised. When *T*'Signed_Zeros is True, a zero result has the sign of *X*. When *Towards* is zero, its sign has no bearing on the result.

50.a **Ramification:** The value of S'Adjacent(0.0, 1.0) is the smallest normalized positive number of the type *T* when *T*'Denorm is False and the smallest denormalized positive number of the type *T* when *T*'Denorm is True.

S'Copy_Sign 51

S'Copy_Sign denotes a function with the following specification:

```
function S'Copy_Sign (Value, Sign : T)
  return T
```
52

If the value of *Value* is nonzero, the function yields a result whose magnitude is that of *Value* and whose sign is that of *Sign*; otherwise, it yields the value zero. Constraint_Error is optionally raised if the result is outside the base range of S. A zero result has the sign of *Sign* when S'Signed_Zeros is True. 53

Discussion: S'Copy_Sign is provided for convenience in restoring the sign to a quantity from which it has been temporarily removed, or to a related quantity. When S'Signed_Zeros is True, it is also instrumental in determining the sign of a zero quantity, when required. (Because negative and positive zeros compare equal in systems conforming to IEC 559:1989, a negative zero does *not* appear to be negative when compared to zero.) The sign determination is accomplished by transferring the sign of the zero quantity to a nonzero quantity and then testing for a negative result. 53.a

S'Leading_Part 54

S'Leading_Part denotes a function with the following specification:

```
function S'Leading_Part (X : T;
                         Radix_Digits : universal_integer)
  return T
```
55

Let v be the value T'Machine_Radix$^{k-Radix_Digits}$, where k is the normalized exponent of X. The function yields the value 56

- $\lfloor X/v \rfloor \cdot v$, when X is nonnegative and *Radix_Digits* is positive; 57
- $\lceil X/v \rceil \cdot v$, when X is negative and *Radix_Digits* is positive. 58

Constraint_Error is raised when *Radix_Digits* is zero or negative. A zero result[, which can only occur when X is zero,] has the sign of X. 59

Discussion: Informally, if X is nonzero, the result is the value obtained by retaining only the specified number of (leading) significant digits of X (in the machine radix), setting all other digits to zero. 59.a

Implementation Note: The result can be obtained by first scaling X up, if necessary to normalize it, then masking the mantissa so as to retain only the specified number of leading digits, then scaling the result back down if X was scaled up. 59.b

S'Machine S'Machine denotes a function with the following specification: 60

```
function S'Machine (X : T)
  return T
```
61

If X is a machine number of the type T, the function yields X; otherwise, it yields the value obtained by rounding or truncating X to either one of the adjacent machine numbers of the type T. Constraint_Error is raised if rounding or truncating X to the precision of the machine numbers results in a value outside the base range of S. A zero result has the sign of X when S'Signed_Zeros is True. 62

Discussion: {*AI05-0005-1*} All of the primitive function attributes except Rounding and Machine correspond to subprograms in the Generic_Primitive_Functions generic package that was proposed as a separate ISO standard (ISO/IEC DIS 11729) for Ada 83. The Scaling, Unbiased_Rounding, and Truncation attributes correspond to the Scale, Round, and Truncate functions, respectively, in Generic_Primitive_Functions. The Rounding attribute rounds away from zero; this functionality was not provided in Generic_Primitive_Functions. The name Round was not available for either of the primitive function attributes that perform rounding, since an attribute of that name is used for a different purpose for decimal fixed point types. Likewise, the name Scale was not available, since an attribute of that name is also used for a different purpose for decimal fixed point types. The functionality of the Machine attribute was also not provided in Generic_Primitive_Functions. The functionality of the Decompose procedure of Generic_Primitive_Functions is only provided in the form of the separate attributes Exponent and Fraction. The functionality of the Successor and Predecessor functions of Generic_Primitive_Functions is provided by the extension of the existing Succ and Pred attributes. 62.a/3

Implementation Note: The primitive function attributes may be implemented either with appropriate floating point arithmetic operations or with integer and logical operations that act on parts of the representation directly. The latter is strongly encouraged when it is more efficient than the former; it is mandatory when the former cannot deliver the required accuracy due to limitations of the implementation's arithmetic operations. 62.b

63 The following *model-oriented attributes* are defined for any subtype S of a floating point type *T*.

64 S'Model_Mantissa

If the Numerics Annex is not supported, this attribute yields an implementation defined value that is greater than or equal to $\lceil d \cdot \log(10) / \log(T\text{'Machine\_Radix}) \rceil + 1$, where *d* is the requested decimal precision of *T*, and less than or equal to the value of *T*'Machine_Mantissa. See G.2.2 for further requirements that apply to implementations supporting the Numerics Annex. The value of this attribute is of the type *universal_integer*.

65 S'Model_Emin

If the Numerics Annex is not supported, this attribute yields an implementation defined value that is greater than or equal to the value of *T*'Machine_Emin. See G.2.2 for further requirements that apply to implementations supporting the Numerics Annex. The value of this attribute is of the type *universal_integer*.

66 S'Model_Epsilon

Yields the value $T\text{'Machine\_Radix}^{1 - T\text{'Model\_Mantissa}}$. The value of this attribute is of the type *universal_real*.

66.a **Discussion:** In most implementations, this attribute yields the absolute value of the difference between one and the smallest machine number of the type *T* above one which, when added to one, yields a machine number different from one. Further discussion can be found in G.2.2.

67 S'Model_Small

Yields the value $T\text{'Machine\_Radix}^{T\text{'Model\_Emin} - 1}$. The value of this attribute is of the type *universal_real*.

67.a/5 **Discussion:** {*AI12-0453-1*} In most implementations, this attribute yields the smallest positive normalized number of the type *T*, that is, the number corresponding to the positive underflow threshold. In some implementations employing a radix-complement representation for the type *T*, the positive underflow threshold is closer to zero than is the negative underflow threshold, with the consequence that the smallest positive normalized number does not coincide with the positive underflow threshold that is, it exceeds the latter). Further discussion can be found in G.2.2.

68 S'Model S'Model denotes a function with the following specification:

69
```
function S'Model (X : T)
  return T
```

70 If the Numerics Annex is not supported, the meaning of this attribute is implementation defined; see G.2.2 for the definition that applies to implementations supporting the Numerics Annex.

71 S'Safe_First

Yields the lower bound of the safe range (see 3.5.7) of the type *T*. If the Numerics Annex is not supported, the value of this attribute is implementation defined; see G.2.2 for the definition that applies to implementations supporting the Numerics Annex. The value of this attribute is of the type *universal_real*.

72 S'Safe_Last

Yields the upper bound of the safe range (see 3.5.7) of the type *T*. If the Numerics Annex is not supported, the value of this attribute is implementation defined; see G.2.2 for the definition that applies to implementations supporting the Numerics Annex. The value of this attribute is of the type *universal_real*.

72.a **Discussion:** A predefined floating point arithmetic operation that yields a value in the safe range of its result type is guaranteed not to overflow.

72.b **To be honest:** An exception is made for exponentiation by a negative exponent in 4.5.6.

72.c **Implementation defined:** The values of the Model_Mantissa, Model_Emin, Model_Epsilon, Model, Safe_First, and Safe_Last attributes, if the Numerics Annex is not supported.

Incompatibilities With Ada 83

The Epsilon and Mantissa attributes of floating point types are removed from the language and replaced by Model_Epsilon and Model_Mantissa, which may have different values (as a result of changes in the definition of model numbers); the replacement of one set of attributes by another is intended to convert what would be an inconsistent change into an incompatible change. 72.d

The Emax, Small, Large, Safe_Emax, Safe_Small, and Safe_Large attributes of floating point types are removed from the language. Small and Safe_Small are collectively replaced by Model_Small, which is functionally equivalent to Safe_Small, though it may have a slightly different value. The others are collectively replaced by Safe_First and Safe_Last. Safe_Last is functionally equivalent to Safe_Large, though it may have a different value; Safe_First is comparable to the negation of Safe_Large but may differ slightly from it as well as from the negation of Safe_Last. Emax and Safe_Emax had relatively few uses in Ada 83; T'Safe_Emax can be computed in the revised language as Integer'Min(T'Exponent(T'Safe_First), T'Exponent(T'Safe_Last)). 72.e

Implementations are encouraged to eliminate the incompatibilities discussed here by retaining the old attributes, during a transition period, in the form of implementation-defined attributes with their former values. 72.f

Extensions to Ada 83

The Model_Emin attribute is new. It is conceptually similar to the negation of Safe_Emax attribute of Ada 83, adjusted for the fact that the model numbers now have the hardware radix. It is a fundamental determinant, along with Model_Mantissa, of the set of model numbers of a type (see G.2.1). 72.g

The Denorm and Signed_Zeros attributes are new, as are all of the primitive function attributes. 72.h

Extensions to Ada 95

{*AI95-00388-01*} The Machine_Rounding attribute is new. 72.i/2

A.5.4 Attributes of Fixed Point Types

Static Semantics

The following *representation-oriented* attributes are defined for every subtype S of a fixed point type *T*. 1

S'Machine_Radix 2

Yields the radix of the hardware representation of the type *T*. The value of this attribute is of the type *universal_integer*.

S'Machine_Rounds 3

Yields the value True if rounding is performed on inexact results of every predefined operation that yields a result of the type *T*; yields the value False otherwise. The value of this attribute is of the predefined type Boolean.

S'Machine_Overflows 4

Yields the value True if overflow and divide-by-zero are detected and reported by raising Constraint_Error for every predefined operation that yields a result of the type *T*; yields the value False otherwise. The value of this attribute is of the predefined type Boolean.

Incompatibilities With Ada 83

The Mantissa, Large, Safe_Small, and Safe_Large attributes of fixed point types are removed from the language. 4.a

Implementations are encouraged to eliminate the resulting incompatibility by retaining these attributes, during a transition period, in the form of implementation-defined attributes with their former values. 4.b

Extensions to Ada 83

The Machine_Radix attribute is now allowed for fixed point types. It is also specifiable in an attribute definition clause (see F.1). 4.c

A.5.5 Big Numbers

1/5 {*AI12-0208-1*} Support is provided for integer arithmetic involving values larger than those supported by the target machine, and for arbitrary-precision real numbers.

Static Semantics

2/5 {*AI12-0208-1*} The library package Numerics.Big_Numbers has the following declaration:

3/5
```
package Ada.Numerics.Big_Numbers
   with Pure, Nonblocking, Global => null is
   subtype Field is Integer range 0 .. implementation-defined;
   subtype Number_Base is Integer range 2 .. 16;
end Ada.Numerics.Big_Numbers;
```

Extensions to Ada 2012

3.a/5 {*AI12-0208-1*} The package Numerics.Big_Numbers is new.

A.5.6 Big Integers

Static Semantics

1/5 {*AI12-0208-1*} {*AI12-0366-1*} The library package Numerics.Big_Numbers.Big_Integers has the following declaration:

2/5
```
with Ada.Strings.Text_Buffers;
package Ada.Numerics.Big_Numbers.Big_Integers
   with Preelaborate, Nonblocking, Global => in out synchronized is
```

3/5
```
   {AI12-0208-1} {AI12-0366-1} {AI12-0407-1}    type Big_Integer is private
        with Integer_Literal => From_Universal_Image,
             Put_Image => Put_Image;
```

4/5
```
   function Is_Valid (Arg : Big_Integer) return Boolean
      with Convention => Intrinsic;
```

4.a/5 **Discussion:** {*AI12-0005-1*} The result of Is_Valid on a default-initialized object of type Big_Integer is unspecified, analogous to the value of a Valid attribute_reference applied to a default-initialized object of an integer type (see 13.9.2). The language-provided functions in the Big_Integers package only return values for which Is_Valid is certain to be True.

5/5
```
   subtype Valid_Big_Integer is Big_Integer
      with Dynamic_Predicate => Is_Valid (Valid_Big_Integer),
           Predicate_Failure => (raise Program_Error);
```

6/5
```
   function "=" (L, R : Valid_Big_Integer) return Boolean;
   function "<" (L, R : Valid_Big_Integer) return Boolean;
   function "<=" (L, R : Valid_Big_Integer) return Boolean;
   function ">" (L, R : Valid_Big_Integer) return Boolean;
   function ">=" (L, R : Valid_Big_Integer) return Boolean;
```

7/5
```
   function To_Big_Integer (Arg : Integer) return Valid_Big_Integer;
```

8/5
```
   subtype Big_Positive is Big_Integer
      with Dynamic_Predicate => (if Is_Valid (Big_Positive)
                                 then Big_Positive > 0),
           Predicate_Failure => (raise Constraint_Error);
```

9/5
```
   subtype Big_Natural is Big_Integer
      with Dynamic_Predicate => (if Is_Valid (Big_Natural)
                                 then Big_Natural >= 0),
           Predicate_Failure => (raise Constraint_Error);
```

10/5
```
   function In_Range (Arg, Low, High : Valid_Big_Integer) return Boolean is
     (Low <= Arg and Arg <= High);
```

```
   function To_Integer (Arg : Valid_Big_Integer) return Integer
      with Pre => In_Range (Arg,
                            Low  => To_Big_Integer (Integer'First),
                            High => To_Big_Integer (Integer'Last))
                  or else raise Constraint_Error;
```
11/5

```
   generic
      type Int is range <>;
   package Signed_Conversions is
      function To_Big_Integer (Arg : Int) return Valid_Big_Integer;
      function From_Big_Integer (Arg : Valid_Big_Integer) return Int
         with Pre => In_Range (Arg,
                               Low  => To_Big_Integer (Int'First),
                               High => To_Big_Integer (Int'Last))
                     or else raise Constraint_Error;
   end Signed_Conversions;
```
12/5

```
   generic
      type Int is mod <>;
   package Unsigned_Conversions is
      function To_Big_Integer (Arg : Int) return Valid_Big_Integer;
      function From_Big_Integer (Arg : Valid_Big_Integer) return Int
         with Pre => In_Range (Arg,
                               Low  => To_Big_Integer (Int'First),
                               High => To_Big_Integer (Int'Last))
                     or else raise Constraint_Error;
   end Unsigned_Conversions;
```
13/5

```
   function To_String (Arg : Valid_Big_Integer;
                       Width : Field := 0;
                       Base  : Number_Base := 10) return String
      with Post => To_String'Result'First = 1;
```
14/5

```
   function From_String (Arg : String) return Valid_Big_Integer;
```
15/5

{*AI12-0407-1*}
```
   function From_Universal_Image (Arg : String) return
Valid_Big_Integer
      renames From_String;
```
16/5

```
   procedure Put_Image
     (Buffer : in out Ada.Strings.Text_Buffers.Root_Buffer_Type'Class;
      Arg    : in Valid_Big_Integer);
```
17/5

```
   function "+" (L : Valid_Big_Integer) return Valid_Big_Integer;
   function "-" (L : Valid_Big_Integer) return Valid_Big_Integer;
   function "abs" (L : Valid_Big_Integer) return Valid_Big_Integer;
   function "+" (L, R : Valid_Big_Integer) return Valid_Big_Integer;
   function "-" (L, R : Valid_Big_Integer) return Valid_Big_Integer;
   function "*" (L, R : Valid_Big_Integer) return Valid_Big_Integer;
   function "/" (L, R : Valid_Big_Integer) return Valid_Big_Integer;
   function "mod" (L, R : Valid_Big_Integer) return Valid_Big_Integer;
   function "rem" (L, R : Valid_Big_Integer) return Valid_Big_Integer;
   function "**" (L : Valid_Big_Integer; R : Natural)
      return Valid_Big_Integer;
   function Min (L, R : Valid_Big_Integer) return Valid_Big_Integer;
   function Max (L, R : Valid_Big_Integer) return Valid_Big_Integer;
```
18/5

```
   function Greatest_Common_Divisor
     (L, R : Valid_Big_Integer) return Big_Positive
     with Pre => (L /= 0 and R /= 0) or else raise Constraint_Error;
```
19/5

```
private
   ... -- not specified by the language
end Ada.Numerics.Big_Numbers.Big_Integers;
```
20/5

{*AI12-0208-1*} {*AI12-0366-1*} To_String and From_String behave analogously to the Put and Get procedures defined in Text_IO.Integer_IO (in particular, with respect to the interpretation of the Width and Base parameters) except that Constraint_Error, not Data_Error, is propagated in error cases and the result of a call to To_String with a Width parameter of 0 and a nonnegative Arg parameter does not include a leading blank. Put_Image calls To_String (passing in the default values for the Width and Base 21/5

parameters), prepends a leading blank if the argument is nonnegative, and writes the resulting value to the buffer using Text_Buffers.Put.

22/5 {*AI12-0208-1*} The other functions have their usual mathematical meanings.

23/5 {*AI12-0208-1*} {*AI12-0366-1*} The type Big_Integer needs finalization (see 7.6).

Dynamic Semantics

24/5 {*AI12-0208-1*} {*AI12-0366-1*} For purposes of determining whether predicate checks are performed as part of default initialization, the type Big_Integer is considered to have a subcomponent that has a default_expression.

24.a/5 **Ramification:** {*AI12-0005-1*} {*AI12-0208-1*} This means that the elaboration of

24.b/5
```
Default_Initialized_Object : Valid_Big_Integer;
```

24.c/5 either produces a value for which Is_Valid is True. or it propagates Program_Error.

Implementation Requirements

25/5 {*AI12-0208-1*} {*AI12-0366-1*} No storage associated with a Big_Integer object shall be lost upon assignment or scope exit.

25.a/5 **Implementation Note:** {*AI12-0208-1*} The "No storage ... shall be lost" requirement does not preclude implementation techniques such as caching or unique number tables.

Extensions to Ada 2012

25.b/5 {*AI12-0208-1*} {*AI12-0366-1*} The package Numerics.Big_Numbers.Big_Integers is new.

A.5.7 Big Reals

Static Semantics

1/5 {*AI12-0208-1*} {*AI12-0366-1*} The library package Numerics.Big_Numbers.Big_Reals has the following declaration:

2/5
```
with Ada.Numerics.Big_Numbers.Big_Integers;
   use all type Big_Integers.Big_Integer;
with Ada.Strings.Text_Buffers;
package Ada.Numerics.Big_Numbers.Big_Reals
   with Preelaborate, Nonblocking, Global => in out synchronized is
```

3/5
```
   {AI12-0208-1} {AI12-0366-1} {AI12-0407-1}    type Big_Real is private
         with Real_Literal => From_Universal_Image,
              Put_Image => Put_Image;
```

4/5
```
      function Is_Valid (Arg : Big_Real) return Boolean
         with Convention => Intrinsic;
```

4.a/5 **Discussion:** {*AI12-0005-1*} The result of Is_Valid on a default-initialized object of type Big_Real is unspecified, analogous to the value of a Valid attribute_reference applied to a default-initialized object of a real type (see 13.9.2). The language-provided functions in the Big_Reals package only return values for which Is_Valid is certain to be True.

5/5
```
      subtype Valid_Big_Real is Big_Real
         with Dynamic_Predicate => Is_Valid (Valid_Big_Real),
              Predicate_Failure => raise Program_Error;
```

6/5
```
      function "/" (Num, Den : Big_Integers.Valid_Big_Integer)
         return Valid_Big_Real
         with Pre => Den /= 0
                     or else raise Constraint_Error;
```

7/5
```
      function Numerator
         (Arg : Valid_Big_Real) return Big_Integers.Valid_Big_Integer
        with Post => (if Arg = 0.0 then Numerator'Result = 0);
```

Reason: The postcondition of Numerator cannot be complete as it cannot mention Denominator. Since the postcondition of Denominator uses Numerator, we would get an infinite mutual recursion if both postconditions are enabled. The postcondition of Denominator serves as the postcondition for Numerator as well unless Arg = 0.0. 7.a/5

```
function Denominator (Arg : Valid_Big_Real)                                    8/5
   return Big_Integers.Big_Positive
   with Post =>
     (if Arg = 0.0 then Denominator'Result = 1
      else Big_Integers.Greatest_Common_Divisor
             (Numerator (Arg), Denominator'Result) = 1);

function To_Big_Real (Arg : Big_Integers.Valid_Big_Integer)                    9/5
   return Valid_Big_Real is (Arg / 1);

function To_Real (Arg : Integer) return Valid_Big_Real is                      10/5
   (Big_Integers.To_Big_Integer (Arg) / 1);

function "=" (L, R : Valid_Big_Real) return Boolean;                           11/5
function "<" (L, R : Valid_Big_Real) return Boolean;
function "<=" (L, R : Valid_Big_Real) return Boolean;
function ">" (L, R : Valid_Big_Real) return Boolean;
function ">=" (L, R : Valid_Big_Real) return Boolean;

function In_Range (Arg, Low, High : Valid_Big_Real) return Boolean is          12/5
   (Low <= Arg and Arg <= High);

generic                                                                        13/5
   type Num is digits <>;
package Float_Conversions is
   function To_Big_Real (Arg : Num) return Valid_Big_Real;
   function From_Big_Real (Arg : Valid_Big_Real) return Num
      with Pre => In_Range (Arg,
                            Low  => To_Big_Real (Num'First),
                            High => To_Big_Real (Num'Last))
                  or else (raise Constraint_Error);
end Float_Conversions;

generic                                                                        14/5
   type Num is delta <>;
package Fixed_Conversions is
   function To_Big_Real (Arg : Num) return Valid_Big_Real;
   function From_Big_Real (Arg : Valid_Big_Real) return Num
      with Pre => In_Range (Arg,
                            Low  => To_Big_Real (Num'First),
                            High => To_Big_Real (Num'Last))
                  or else (raise Constraint_Error);
end Fixed_Conversions;

function To_String (Arg  : Valid_Big_Real;                                     15/5
                    Fore : Field := 2;
                    Aft  : Field := 3;
                    Exp  : Field := 0) return String
   with Post => To_String'Result'First = 1;

function From_String (Arg   : String) return Valid_Big_Real;                   16/5

{AI12-0407-1}    function From_Universal_Image (Arg : String) return Valid_Big_Real   17/5
   renames From_String;

{AI12-0407-1}    function From_Universal_Image (Num, Den : String)             18/5
   return Valid_Big_Real is
      (Big_Integers.From_Universal_Image (Num) /
       Big_Integers.From_Universal_Image (Den));

function To_Quotient_String (Arg : Valid_Big_Real) return String is            19/5
   (To_String (Numerator (Arg)) & " / " & To_String (Denominator (Arg)));
function From_Quotient_String (Arg : String) return Valid_Big_Real;

procedure Put_Image                                                            20/5
  (Buffer : in out Ada.Strings.Text_Buffers.Root_Buffer_Type'Class;
   Arg    : in Valid_Big_Real);
```

21/5
```
      function "+" (L : Valid_Big_Real) return Valid_Big_Real;
      function "-" (L : Valid_Big_Real) return Valid_Big_Real;
      function "abs" (L : Valid_Big_Real) return Valid_Big_Real;
      function "+" (L, R : Valid_Big_Real) return Valid_Big_Real;
      function "-" (L, R : Valid_Big_Real) return Valid_Big_Real;
      function "*" (L, R : Valid_Big_Real) return Valid_Big_Real;
      function "/" (L, R : Valid_Big_Real) return Valid_Big_Real;
      function "**" (L : Valid_Big_Real; R : Integer)
         return Valid_Big_Real;
      function Min (L, R : Valid_Big_Real) return Valid_Big_Real;
      function Max (L, R : Valid_Big_Real) return Valid_Big_Real;
```

22/5
```
   private
      ... -- not specified by the language
   end Ada.Numerics.Big_Numbers.Big_Reals;
```

23/5 {*AI12-0208-1*} {*AI12-0366-1*} To_String and From_String behave analogously to the Put and Get procedures defined in Text_IO.Float_IO (in particular, with respect to the interpretation of the Fore, Aft, and Exp parameters), except that Constraint_Error (not Data_Error) is propagated in error cases. From_Quotient_String implements the inverse function of To_Quotient_String; Constraint_Error is propagated in error cases. Put_Image calls To_String, and writes the resulting value to the buffer using Text_Buffers.Put.

24/5 {*AI12-0208-1*} For an instance of Float_Conversions or Fixed_Conversions, To_Big_Real is exact (that is, the result represents exactly the same mathematical value as the argument) and From_Big_Real is subject to the same precision rules as a type conversion of a value of type T to the target type Num, where T is a hypothetical floating point type whose model numbers include all of the model numbers of Num as well as the exact mathematical value of the argument.

25/5 {*AI12-0208-1*} The other functions have their usual mathematical meanings.

26/5 {*AI12-0208-1*} {*AI12-0366-1*} The type Big_Real needs finalization (see 7.6).

Dynamic Semantics

27/5 {*AI12-0208-1*} {*AI12-0366-1*} For purposes of determining whether predicate checks are performed as part of default initialization, the type Big_Real is considered to have a subcomponent that has a default_expression.

27.a/5 **Ramification:** {*AI12-0005-1*} {*AI12-0208-1*} This means that the elaboration of

27.b/5
```
Default_Initialized_Object : Valid_Big_Real;
```

27.c/5 either produces a value for which Is_Valid is True. or it propagates Program_Error.

Implementation Requirements

28/5 {*AI12-0208-1*} {*AI12-0366-1*} No storage associated with a Big_Real object shall be lost upon assignment or scope exit.

28.a/5 **Implementation Note:** {*AI12-0208-1*} The “No storage ... shall be lost” requirement does not preclude implementation techniques such as caching or unique number tables.

Extensions to Ada 2012

28.b/5 {*AI12-0208-1*} {*AI12-0366-1*} The package Numerics.Big_Numbers.Big_Reals is new.

A.6 Input-Output

1/5 {*AI95-00285-01*} {*AI12-0445-1*} [Input-output is provided through language-defined packages, each of which is a child of the root package Ada. The generic packages Sequential_IO and Direct_IO define input-output operations applicable to files containing elements of a given type. The generic package Storage_IO supports reading from and writing to an in-memory buffer. Additional operations for text

input-output are supplied in the packages Text_IO, Wide_Text_IO, and Wide_Wide_Text_IO. Heterogeneous input-output is provided through the child packages Streams.Stream_IO and Text_IO.-Text_Streams (see also 13.13). The package IO_Exceptions defines the exceptions used by the predefined input-output packages.]

Inconsistencies With Ada 83

1.a The introduction of Append_File as a new element of the enumeration type File_Mode in Sequential_IO and Text_IO, and the introduction of several new declarations in Text_IO, may result in name clashes in the presence of **use** clauses.

Extensions to Ada 83

1.b Text_IO enhancements (Get_Immediate, Look_Ahead, Standard_Error, Modular_IO, Decimal_IO), Wide_Text_IO, and the stream input-output facilities are new in Ada 95.

Wording Changes from Ada 83

1.c RM83-14.6, “Low Level Input-Output”, is removed. This has no semantic effect, since the package was entirely implementation defined, nobody actually implemented it, and if they did, they can always provide it as a vendor-supplied package.

Wording Changes from Ada 95

1.d/2 {*AI95-00285-01*} Included package Wide_Wide_Text_IO in this description.

A.7 External Files and File Objects

Static Semantics

1 Values input from the external environment of the program, or output to the external environment, are considered to occupy *external files*. An external file can be anything external to the program that can produce a value to be read or receive a value to be written. An external file is identified by a string (the *name*). A second string (the *form*) gives further system-dependent characteristics that may be associated with the file, such as the physical organization or access rights. The conventions governing the interpretation of such strings shall be documented.

2/3 {*AI05-0299-1*} Input and output operations are expressed as operations on objects of some *file type*, rather than directly in terms of the external files. In the remainder of this clause, the term *file* is always used to refer to a file object; the term *external file* is used otherwise.

3 Input-output for sequential files of values of a single element type is defined by means of the generic package Sequential_IO. In order to define sequential input-output for a given element type, an instantiation of this generic unit, with the given type as actual parameter, has to be declared. The resulting package contains the declaration of a file type (called File_Type) for files of such elements, as well as the operations applicable to these files, such as the Open, Read, and Write procedures.

4/2 {*AI95-00285-01*} Input-output for direct access files is likewise defined by a generic package called Direct_IO. Input-output in human-readable form is defined by the (nongeneric) packages Text_IO for Character and String data, Wide_Text_IO for Wide_Character and Wide_String data, and Wide_Wide_Text_IO for Wide_Wide_Character and Wide_Wide_String data. Input-output for files containing streams of elements representing values of possibly different types is defined by means of the (nongeneric) package Streams.Stream_IO.

5 Before input or output operations can be performed on a file, the file first has to be associated with an external file. While such an association is in effect, the file is said to be *open*, and otherwise the file is said to be *closed*.

6 The language does not define what happens to external files after the completion of the main program and all the library tasks (in particular, if corresponding files have not been closed). The effect of input-output for access types is unspecified.

7 An open file has a *current mode*, which is a value of one of the following enumeration types:

8
```
type File_Mode is (In_File, Inout_File, Out_File);  -- for Direct_IO
```

9 These values correspond respectively to the cases where only reading, both reading and writing, or only writing are to be performed.

10/2
```
{AI95-00285-01} type File_Mode is (In_File, Out_File, Append_File);
-- for Sequential_IO, Text_IO, Wide_Text_IO, Wide_Wide_Text_IO, and Stream_IO
```

11 These values correspond respectively to the cases where only reading, only writing, or only appending are to be performed.

12 The mode of a file can be changed.

13/5 {*AI95-00285-01*} {*AI12-0449-1*} Several file management operations are common to Sequential_IO, Direct_IO, Text_IO, Wide_Text_IO, and Wide_Wide_Text_IO. These operations are described in A.8.2 for sequential and direct files. Any additional effects concerning text input-output are described in A.10.2.

14/5 {*AI05-0299-1*} {*AI12-0449-1*} The exceptions that can be propagated by the execution of an input-output subprogram are defined in the package IO_Exceptions; the situations in which they can be propagated are described following the description of the subprogram (and in A.13). The exceptions Storage_Error and Program_Error may be propagated. (Program_Error can only be propagated due to errors made by the caller of the subprogram.) Finally, exceptions can be propagated in certain implementation-defined situations.

14.a/2 *This paragraph was deleted.*

14.b/2 **Discussion:** The last sentence here is referring to the documentation requirements in A.13, “Exceptions in Input-Output”, and the documentation summary item is provided there.

15/2 NOTE 1 {*AI95-00285-01*} Each instantiation of the generic packages Sequential_IO and Direct_IO declares a different type File_Type. In the case of Text_IO, Wide_Text_IO, Wide_Wide_Text_IO, and Streams.Stream_IO, the corresponding type File_Type is unique.

16/5 NOTE 2 {*AI12-0440-1*} A bidirectional device can often be modeled as two sequential files associated with the device, one of mode In_File, and one of mode Out_File. An implementation can restrict the number of files that can be associated with a given external file.

Wording Changes from Ada 95

16.a/2 {*AI95-00285-01*} Included package Wide_Wide_Text_IO in this description.

A.8 Sequential and Direct Files

Static Semantics

1/2 {*AI95-00283-01*} Two kinds of access to external files are defined in this subclause: *sequential access* and *direct access*. The corresponding file types and the associated operations are provided by the generic packages Sequential_IO and Direct_IO. A file object to be used for sequential access is called a *sequential file*, and one to be used for direct access is called a *direct file*. Access to *stream file*s is described in A.12.1.

2 For sequential access, the file is viewed as a sequence of values that are transferred in the order of their appearance (as produced by the program or by the external environment). When the file is opened with mode In_File or Out_File, transfer starts respectively from or to the beginning of the file. When the file is opened with mode Append_File, transfer to the file starts after the last element of the file.

Discussion: Adding stream I/O necessitates a review of the terminology. In Ada 83, `sequential' implies both the access method (purely sequential — that is, no indexing or positional access) and homogeneity. Direct access includes purely sequential access and indexed access, as well as homogeneity. In Ada 95, streams allow purely sequential access but also positional access to an individual element, and are heterogeneous. We considered generalizing the notion of `sequential file' to include both Sequential_IO and Stream_IO files, but since streams allow positional access it seems misleading to call them sequential files. Or, looked at differently, if the criterion for calling something a sequential file is whether it permits (versus requires) purely sequential access, then one could just as soon regard a Direct_IO file as a sequential file. 2.a

It seems better to regard `sequential file' as meaning `only permitting purely sequential access'; hence we have decided to supplement `sequential access' and `direct access' with a third category, informally called `access to streams'. (We decided against the term `stream access' because of possible confusion with the Stream_Access type declared in one of the stream packages.) 2.b

For direct access, the file is viewed as a set of elements occupying consecutive positions in linear order; a value can be transferred to or from an element of the file at any selected position. The position of an element is specified by its *index*, which is a number, greater than zero, of the implementation-defined integer type Count. The first element, if any, has index one; the index of the last element, if any, is called the *current size*; the current size is zero if there are no elements. The current size is a property of the external file. 3

An open direct file has a *current index*, which is the index that will be used by the next read or write operation. When a direct file is opened, the current index is set to one. The current index of a direct file is a property of a file object, not of an external file. 4

Wording Changes from Ada 95

{*AI95-00283-01*} Italicized “stream file” to clarify that this is another kind of file. 4.a/2

A.8.1 The Generic Package Sequential_IO

Static Semantics

The generic library package Sequential_IO has the following declaration: 1

```
{AI12-0302-1} with Ada.IO_Exceptions;                                                    2/5
generic
   type Element_Type(<>) is private;
package Ada.Sequential_IO
   with Global => in out synchronized is

   type File_Type is limited private;                                                    3

   type File_Mode is (In_File, Out_File, Append_File);                                   4

   -- File management                                                                    5

   procedure Create(File : in out File_Type;                                             6
                    Mode : in File_Mode := Out_File;
                    Name : in String := "";
                    Form : in String := "");

   procedure Open  (File : in out File_Type;                                             7
                    Mode : in File_Mode;
                    Name : in String;
                    Form : in String := "");

   procedure Close (File : in out File_Type);                                            8
   procedure Delete(File : in out File_Type);
   procedure Reset (File : in out File_Type; Mode : in File_Mode);
   procedure Reset (File : in out File_Type);

   function Mode   (File : in File_Type) return File_Mode;                               9
   function Name   (File : in File_Type) return String;
   function Form   (File : in File_Type) return String;

   function Is_Open(File : in File_Type) return Boolean;                                 10
```

```
10.1/5      {AI12-0130-1} {AI12-0302-1}    procedure Flush (File : in File_Type)
                with Global => overriding in out File;
11             -- Input and output operations
12/5        {AI12-0302-1}    procedure Read  (File : in File_Type; Item : out Element_Type)
                with Global => overriding in out File;
            procedure Write (File : in File_Type; Item : in Element_Type)
                with Global => overriding in out File;
13          function End_Of_File(File : in File_Type) return Boolean;
14          -- Exceptions
15          Status_Error : exception renames IO_Exceptions.Status_Error;
            Mode_Error   : exception renames IO_Exceptions.Mode_Error;
            Name_Error   : exception renames IO_Exceptions.Name_Error;
            Use_Error    : exception renames IO_Exceptions.Use_Error;
            Device_Error : exception renames IO_Exceptions.Device_Error;
            End_Error    : exception renames IO_Exceptions.End_Error;
            Data_Error   : exception renames IO_Exceptions.Data_Error;
15.1/5      {AI12-0021-1}    package Wide_File_Names is
15.2/5            -- File management
15.3/5            procedure Create(File : in out File_Type;
                                   Mode : in File_Mode := Out_File;
                                   Name : in Wide_String := "";
                                   Form : in Wide_String := "");
15.4/5            procedure Open  (File : in out File_Type;
                                   Mode : in File_Mode;
                                   Name : in Wide_String;
                                   Form : in Wide_String := "");
15.5/5            function Name   (File : in File_Type) return Wide_String;
15.6/5            function Form   (File : in File_Type) return Wide_String;
15.7/5         end Wide_File_Names;
15.8/5      {AI12-0021-1}    package Wide_Wide_File_Names is
15.9/5            -- File management
15.10/5           procedure Create(File : in out File_Type;
                                   Mode : in File_Mode := Out_File;
                                   Name : in Wide_Wide_String := "";
                                   Form : in Wide_Wide_String := "");
15.11/5           procedure Open  (File : in out File_Type;
                                   Mode : in File_Mode;
                                   Name : in Wide_Wide_String;
                                   Form : in Wide_Wide_String := "");
15.12/5           function Name   (File : in File_Type) return Wide_Wide_String;
15.13/5           function Form   (File : in File_Type) return Wide_Wide_String;
15.14/5        end Wide_Wide_File_Names;
16          private
               ... -- not specified by the language
            end Ada.Sequential_IO;
```

17/2 {*AI95-00360-01*} The type File_Type needs finalization (see 7.6) in every instantiation of Sequential_IO.

Incompatibilities With Ada 83

17.a The new enumeration element Append_File may introduce upward incompatibilities. It is possible that a program based on the assumption that File_Mode'Last = Out_File will be illegal (e.g., case statement choice coverage) or execute with a different effect in Ada 95.

17.a.1/2 *This paragraph was deleted.* {*8652/0097*} {*AI95-00115-01*} {*AI95-00344-01*}

Incompatibilities With Ada 95

{*AI95-00360-01*} **Amendment Correction:** File_Type in an instance of Sequential_IO is defined to need finalization. If the restriction No_Nested_Finalization (see D.7) applies to the partition, and File_Type does not have a controlled part, it will not be allowed in local objects in Ada 2005 whereas it would be allowed in original Ada 95. Such code is not portable, as another Ada compiler may have a controlled part in File_Type, and thus would be illegal. 17.b/2

Incompatibilities With Ada 2012

{*AI12-0130-1*} {*AI05-0005-1*} **Corrigendum:** The Flush procedure is newly added to Ada.Sequential_IO. Therefore, a use clause conflict is possible; see the introduction of Annex A for more on this topic. 17.c/5

{*AI12-0021-1*} The Wide_File_Names and Wide_Wide_File_Names nested packages are newly added to Ada.Sequential_IO. Therefore, a use clause conflict is possible; see the introduction of Annex A for more on this topic. 17.d/5

A.8.2 File Management

Static Semantics

{*AI12-0449-1*} The procedures and functions described in this subclause provide for the control of external files; their declarations are repeated in each of the packages for sequential, direct, text, and stream input-output. For text input-output, the procedures Create, Open, and Reset have additional effects described in A.10.2. 1/5

```
procedure Create(File : in out File_Type;                                    2
                 Mode : in File_Mode := default_mode;
                 Name : in String := "";
                 Form : in String := "");
```

{*AI95-00283-01*} Establishes a new external file, with the given name and form, and associates this external file with the given file. The given file is left open. The current mode of the given file is set to the given access mode. The default access mode is the mode Out_File for sequential, stream, and text input-output; it is the mode Inout_File for direct input-output. For direct access, the size of the created file is implementation defined. 3/2

Ramification: {*AI12-0005-1*} For a direct file, the initial value of current index is 1 (see A.8). 3.a/4

A null string for Name specifies an external file that is not accessible after the completion of the main program (a temporary file). A null string for Form specifies the use of the default options of the implementation for the external file. 4

The exception Status_Error is propagated if the given file is already open. The exception Name_Error is propagated if the string given as Name does not allow the identification of an external file. The exception Use_Error is propagated if, for the specified mode, the external environment does not support creation of an external file with the given name (in the absence of Name_Error) and form. 5

```
procedure Open(File : in out File_Type;                                      6
               Mode : in File_Mode;
               Name : in String;
               Form : in String := "");
```

Associates the given file with an existing external file having the given name and form, and sets the current mode of the given file to the given mode. The given file is left open. 7

Ramification: {*AI12-0005-1*} For a direct file, the initial value of current index is 1 (see A.8). 7.a/4

The exception Status_Error is propagated if the given file is already open. The exception Name_Error is propagated if the string given as Name does not allow the identification of an external file; in particular, this exception is propagated if no external file with the given name exists. The exception Use_Error is propagated if, for the specified mode, the external 8

environment does not support opening for an external file with the given name (in the absence of Name_Error) and form.

9 `procedure Close(File : in out File_Type);`

10 Severs the association between the given file and its associated external file. The given file is left closed. In addition, for sequential files, if the file being closed has mode Out_File or Append_File, then the last element written since the most recent open or reset is the last element that can be read from the file. If no elements have been written and the file mode is Out_File, then the closed file is empty. If no elements have been written and the file mode is Append_File, then the closed file is unchanged.

11 The exception Status_Error is propagated if the given file is not open.

12 `procedure Delete(File : in out File_Type);`

13 Deletes the external file associated with the given file. The given file is closed, and the external file ceases to exist.

14 The exception Status_Error is propagated if the given file is not open. The exception Use_Error is propagated if deletion of the external file is not supported by the external environment.

15
```
procedure Reset(File : in out File_Type; Mode : in File_Mode);
procedure Reset(File : in out File_Type);
```

16/2 {*AI95-00085-01*} Resets the given file so that reading from its elements can be restarted from the beginning of the external file (for modes In_File and Inout_File), and so that writing to its elements can be restarted at the beginning of the external file (for modes Out_File and Inout_File) or after the last element of the external file (for mode Append_File). In particular, for direct access this means that the current index is set to one. If a Mode parameter is supplied, the current mode of the given file is set to the given mode. In addition, for sequential files, if the given file has mode Out_File or Append_File when Reset is called, the last element written since the most recent open or reset is the last element that can be read from the external file. If no elements have been written and the file mode is Out_File, the reset file is empty. If no elements have been written and the file mode is Append_File, then the reset file is unchanged.

17 The exception Status_Error is propagated if the file is not open. The exception Use_Error is propagated if the external environment does not support resetting for the external file and, also, if the external environment does not support resetting to the specified mode for the external file.

18 `function Mode(File : in File_Type) return File_Mode;`

19 Returns the current mode of the given file.

20 The exception Status_Error is propagated if the file is not open.

21 `function Name(File : in File_Type) return String;`

22/2 {*AI95-00248-01*} Returns a string which uniquely identifies the external file currently associated with the given file (and may thus be used in an Open operation).

22.a/2 **Discussion:** {*AI95-00248-01*} Retrieving the full path can be accomplished by passing the result of Name to Directories.Full_Name (see A.16). It is important to drop the requirement on Name, as the only way to accomplish this requirement given that the current directory can be changed with package Directories is to store the full path when the file is opened. That's expensive, and it's better for users that need the full path to explicitly request it.

23 The exception Status_Error is propagated if the given file is not open. The exception Use_Error is propagated if the associated external file is a temporary file that cannot be opened by any name.

```
function Form(File : in File_Type) return String;
```

Returns the form string for the external file currently associated with the given file. If an external environment allows alternative specifications of the form (for example, abbreviations using default options), the string returned by the function should correspond to a full specification (that is, it should indicate explicitly all options selected, including default options).

The exception Status_Error is propagated if the given file is not open.

```
function Is_Open(File : in File_Type) return Boolean;
```

{*AI05-0264-1*} Returns True if the file is open (that is, if it is associated with an external file); otherwise, returns False.

{*AI12-0130-1*}
```
procedure Flush(File : in File_Type);
```

The Flush procedure synchronizes the external file with the internal file (by flushing any internal buffers) without closing the file. For a direct file, the current index is unchanged; for a stream file (see A.12.1), the current position is unchanged.

The exception Status_Error is propagated if the file is not open. The exception Mode_Error is propagated if the mode of the file is In_File.

{*AI12-0021-1*} The nested package Wide_File_Names provides operations equivalent to the operations of the same name of the outer package except that Wide_String is used instead of String for the name and form of the external file.

Implementation Note: {*AI12-0021-1*} The expectation is that if one opens a file with this subpackage and then requests the name with Name from this subpackage, the result will include the same Wide_Characters that were used to open the file (rather than some encoded form). Since file names are completely implementation-defined, we cannot say this normatively. We have no expectations when retrieving the name of a file opened with this package using the String Name. Similar considerations apply to the nested package Wide_Wide_File_Names (see below).

{*AI12-0021-1*} The nested package Wide_Wide_File_Names provides operations equivalent to the operations of the same name of the outer package except that Wide_Wide_String is used instead of String for the name and form of the external file.

Implementation Permissions

An implementation may propagate Name_Error or Use_Error if an attempt is made to use an I/O feature that cannot be supported by the implementation due to limitations in the external environment. Any such restriction should be documented.

Wording Changes from Ada 95

{*AI95-00085-01*} Clarified that Reset affects and depends on the external file.

{*AI95-00248-01*} Removed the requirement for Name to return a full path; this is now accomplished by Directories.Full_Name(Name(File)) (see A.16). This is not documented as an inconsistency, because there is no requirement for implementations to change — the Ada 95 behavior is still allowed, it just is no longer required.

{*AI95-00283-01*} Added text to specify the default mode for a stream file.

Wording Changes from Ada 2012

{*AI12-0130-1*} Procedure Flush is now defined here, so it can be used for all of the I/O packages.

{*AI12-0021-1*} The nested Wide_File_Names and Wide_Wide_File_Names packages are now described here.

A.8.3 Sequential Input-Output Operations

Static Semantics

1 The operations available for sequential input and output are described in this subclause. The exception Status_Error is propagated if any of these operations is attempted for a file that is not open.

2
```
procedure Read(File : in File_Type; Item : out Element_Type);
```

3 Operates on a file of mode In_File. Reads an element from the given file, and returns the value of this element in the Item parameter.

3.a **Discussion:** We considered basing Sequential_IO.Read on Element_Type'Read from an implicit stream associated with the sequential file. However, Element_Type'Read is a type-related attribute, whereas Sequential_IO should take advantage of the particular constraints of the actual subtype corresponding to Element_Type to minimize the size of the external file. Furthermore, forcing the implementation of Sequential_IO to be based on Element_Type'Read would create an upward incompatibility since existing data files written by an Ada 83 program using Sequential_IO might not be readable by the identical program built with an Ada 95 implementation of Sequential_IO.

3.b An Ada 95 implementation might still use an implementation-defined attribute analogous to 'Read to implement the procedure Read, but that attribute will likely have to be subtype-specific rather than type-related, and it need not be user-specifiable. Such an attribute will presumably be needed to implement the generic package Storage_IO (see A.9).

4 The exception Mode_Error is propagated if the mode is not In_File. The exception End_Error is propagated if no more elements can be read from the given file. The exception Data_Error can be propagated if the element read cannot be interpreted as a value of the subtype Element_Type (see A.13, “Exceptions in Input-Output”).

4.a **Discussion:** Data_Error need not be propagated if the check is too complex. See A.13, “Exceptions in Input-Output”.

5
```
procedure Write(File : in File_Type; Item : in Element_Type);
```

6 Operates on a file of mode Out_File or Append_File. Writes the value of Item to the given file.

7 The exception Mode_Error is propagated if the mode is not Out_File or Append_File. The exception Use_Error is propagated if the capacity of the external file is exceeded.

8
```
function End_Of_File(File : in File_Type) return Boolean;
```

9/3 {*AI05-0264-1*} Operates on a file of mode In_File. Returns True if no more elements can be read from the given file; otherwise, returns False.

10 The exception Mode_Error is propagated if the mode is not In_File.

A.8.4 The Generic Package Direct_IO

Static Semantics

1 The generic library package Direct_IO has the following declaration:

2/5
```
{AI12-0302-1} with Ada.IO_Exceptions;
generic
   type Element_Type is private;
package Ada.Direct_IO
   with Global => in out synchronized is

3     type File_Type is limited private;

4     type File_Mode is (In_File, Inout_File, Out_File);
      type Count     is range 0 .. implementation-defined;
      subtype Positive_Count is Count range 1 .. Count'Last;

5     -- File management
```

```
   procedure Create(File : in out File_Type;                                    6
                    Mode : in File_Mode := Inout_File;
                    Name : in String := "";
                    Form : in String := "");

   procedure Open  (File : in out File_Type;                                    7
                    Mode : in File_Mode;
                    Name : in String;
                    Form : in String := "");

   procedure Close (File : in out File_Type);                                   8
   procedure Delete(File : in out File_Type);
   procedure Reset (File : in out File_Type; Mode : in File_Mode);
   procedure Reset (File : in out File_Type);

   function Mode   (File : in File_Type) return File_Mode;                      9
   function Name   (File : in File_Type) return String;
   function Form   (File : in File_Type) return String;

   function Is_Open(File : in File_Type) return Boolean;                        10

{AI12-0130-1} {AI12-0302-1}    procedure Flush (File : in File_Type)          10.1/5
      with Global => overriding in out File;

   -- Input and output operations                                             11

{AI12-0302-1}    procedure Read (File : in File_Type; Item : out Element_Type; 12/5
                                 From : in Positive_Count)
      with Global => overriding in out File;
   procedure Read (File : in File_Type; Item : out Element_Type)
      with Global => overriding in out File;

{AI12-0302-1}    procedure Write(File : in File_Type; Item : in  Element_Type; 13/5
                                 To   : in Positive_Count)
      with Global => overriding in out File;
   procedure Write(File : in File_Type; Item : in Element_Type)
      with Global => overriding in out File;

{AI12-0302-1}    procedure Set_Index(File : in File_Type; To : in Positive_Count) 14/5
      with Global => overriding in out File;

   function Index(File : in File_Type) return Positive_Count;                   15
   function Size (File : in File_Type) return Count;

   function End_Of_File(File : in File_Type) return Boolean;                    16

   -- Exceptions                                                              17

   Status_Error : exception renames IO_Exceptions.Status_Error;                 18
   Mode_Error   : exception renames IO_Exceptions.Mode_Error;
   Name_Error   : exception renames IO_Exceptions.Name_Error;
   Use_Error    : exception renames IO_Exceptions.Use_Error;
   Device_Error : exception renames IO_Exceptions.Device_Error;
   End_Error    : exception renames IO_Exceptions.End_Error;
   Data_Error   : exception renames IO_Exceptions.Data_Error;

{AI12-0021-1}    package Wide_File_Names is                                  18.1/5

      -- File management                                                      18.2/5

      procedure Create(File : in out File_Type;                              18.3/5
                       Mode : in File_Mode := Inout_File;
                       Name : in Wide_String := "";
                       Form : in Wide_String := "");

      procedure Open  (File : in out File_Type;                              18.4/5
                       Mode : in File_Mode;
                       Name : in Wide_String;
                       Form : in Wide_String := "");

      function Name   (File : in File_Type) return Wide_String;              18.5/5

      function Form   (File : in File_Type) return Wide_String;              18.6/5

   end Wide_File_Names;                                                      18.7/5

{AI12-0021-1}    package Wide_Wide_File_Names is                             18.8/5
```

18.9/5
```
      -- File management
```
18.10/5
```
      procedure Create(File : in out File_Type;
                       Mode : in File_Mode := Inout_File;
                       Name : in Wide_Wide_String := "";
                       Form : in Wide_Wide_String := "");
```
18.11/5
```
      procedure Open  (File : in out File_Type;
                       Mode : in File_Mode;
                       Name : in Wide_Wide_String;
                       Form : in Wide_Wide_String := "");
```
18.12/5
```
      function Name   (File : in File_Type) return Wide_Wide_String;
```
18.13/5
```
      function Form   (File : in File_Type) return Wide_Wide_String;
```
18.14/5
```
   end Wide_Wide_File_Names;
```
19
```
private
   ... -- not specified by the language
end Ada.Direct_IO;
```

19.a **Reason:** The Element_Type formal of Direct_IO does not have an unknown_discriminant_part (unlike Sequential_IO) so that the implementation can make use of the ability to declare uninitialized variables of the type.

20/2 {*AI95-00360-01*} The type File_Type needs finalization (see 7.6) in every instantiation of Direct_IO.

20.a.1/2 *This paragraph was deleted.*{*8652/0097*} {*AI95-00115-01*} {*AI95-00344-01*}

Incompatibilities With Ada 95

20.a/2 {*AI95-00360-01*} **Amendment Correction:** File_Type in an instance of Direct_IO is defined to need finalization. If the restriction No_Nested_Finalization (see D.7) applies to the partition, and File_Type does not have a controlled part, it will not be allowed in local objects in Ada 2005 whereas it would be allowed in original Ada 95. Such code is not portable, as another Ada compiler may have a controlled part in File_Type, and thus would be illegal.

Incompatibilities With Ada 2012

20.b/5 {*AI12-0130-1*} {*AI05-0005-1*} **Corrigendum:** The Flush procedure is newly added to Ada.Direct_IO. Therefore, a use clause conflict is possible; see the introduction of Annex A for more on this topic.

20.c/5 {*AI12-0021-1*} The Wide_File_Names and Wide_Wide_File_Names nested packages are newly added to Ada.Direct_IO. Therefore, a use clause conflict is possible; see the introduction of Annex A for more on this topic.

A.8.5 Direct Input-Output Operations

Static Semantics

1 The operations available for direct input and output are described in this subclause. The exception Status_Error is propagated if any of these operations is attempted for a file that is not open.

2
```
   procedure Read(File : in File_Type; Item : out Element_Type;
                                       From : in  Positive_Count);
   procedure Read(File : in File_Type; Item : out Element_Type);
```

3 Operates on a file of mode In_File or Inout_File. In the case of the first form, sets the current index of the given file to the index value given by the parameter From. Then (for both forms) returns, in the parameter Item, the value of the element whose position in the given file is specified by the current index of the file; finally, increases the current index by one.

4 The exception Mode_Error is propagated if the mode of the given file is Out_File. The exception End_Error is propagated if the index to be used exceeds the size of the external file. The exception Data_Error can be propagated if the element read cannot be interpreted as a value of the subtype Element_Type (see A.13).

```
procedure Write(File : in File_Type; Item : in Element_Type;                    5
                                     To   : in Positive_Count);
procedure Write(File : in File_Type; Item : in Element_Type);
```

Operates on a file of mode Inout_File or Out_File. In the case of the first form, sets the index of the given file to the index value given by the parameter To. Then (for both forms) gives the value of the parameter Item to the element whose position in the given file is specified by the current index of the file; finally, increases the current index by one. 6

The exception Mode_Error is propagated if the mode of the given file is In_File. The exception Use_Error is propagated if the capacity of the external file is exceeded. 7

```
procedure Set_Index(File : in File_Type; To : in Positive_Count);               8
```

Operates on a file of any mode. Sets the current index of the given file to the given index value (which may exceed the current size of the file). 9

```
function Index(File : in File_Type) return Positive_Count;                      10
```

Operates on a file of any mode. Returns the current index of the given file. 11

```
function Size(File : in File_Type) return Count;                                12
```

Operates on a file of any mode. Returns the current size of the external file that is associated with the given file. 13

```
function End_Of_File(File : in File_Type) return Boolean;                       14
```

{*AI05-0264-1*} Operates on a file of mode In_File or Inout_File. Returns True if the current index exceeds the size of the external file; otherwise, returns False. 15/3

The exception Mode_Error is propagated if the mode of the given file is Out_File. 16

NOTE Append_File mode is not supported for the generic package Direct_IO. 17

A.9 The Generic Package Storage_IO

The generic package Storage_IO provides for reading from and writing to an in-memory buffer. This generic package supports the construction of user-defined input-output packages. 1

Reason: This package exists to allow the portable construction of user-defined direct-access-oriented input-output packages. The Write procedure writes a value of type Element_Type into a Storage_Array of size Buffer_Size, flattening out any implicit levels of indirection used in the representation of the type. The Read procedure reads a value of type Element_Type from the buffer, reconstructing any implicit levels of indirection used in the representation of the type. It also properly initializes any type tags that appear within the value, presuming that the buffer was written by a different program and that tag values for the"same" type might vary from one executable to another. 1.a

Static Semantics

The generic library package Storage_IO has the following declaration: 2

```
{AI12-0302-1} with Ada.IO_Exceptions;                                           3/5
with System.Storage_Elements;
generic
   type Element_Type is private;
package Ada.Storage_IO
   with Preelaborate, Global => in out synchronized is

   Buffer_Size : constant System.Storage_Elements.Storage_Count :=              4
      implementation-defined;
   subtype Buffer_Type is
      System.Storage_Elements.Storage_Array(1..Buffer_Size);

   -- Input and output operations                                              5

   procedure Read (Buffer : in  Buffer_Type; Item : out Element_Type);          6
```

```
7          procedure Write(Buffer : out Buffer_Type; Item : in  Element_Type);
8          -- Exceptions
9          Data_Error   : exception renames IO_Exceptions.Data_Error;
       end Ada.Storage_IO;
```

10 In each instance, the constant Buffer_Size has a value that is the size (in storage elements) of the buffer required to represent the content of an object of subtype Element_Type, including any implicit levels of indirection used by the implementation. The Read and Write procedures of Storage_IO correspond to the Read and Write procedures of Direct_IO (see A.8.4), but with the content of the Item parameter being read from or written into the specified Buffer, rather than an external file.

10.a **Reason:** As with Direct_IO, the Element_Type formal of Storage_IO does not have an unknown_discriminant_part so that there is a well-defined upper bound on the size of the buffer needed to hold the content of an object of the formal subtype (i.e. Buffer_Size). If there are no implicit levels of indirection, Buffer_Size will typically equal:

10.b `(Element_Type'Size + System.Storage_Unit - 1) / System.Storage_Unit`

10.c **Implementation defined:** The value of Buffer_Size in Storage_IO.

11 NOTE A buffer used for Storage_IO holds only one element at a time; an external file used for Direct_IO holds a sequence of elements.

Extensions to Ada 83

11.a/3 {*AI05-0005-1*} Storage_IO is new in Ada 95.

A.10 Text Input-Output

Static Semantics

1/5 {*AI05-0299-1*} {*AI12-0449-1*} This subclause describes the package Text_IO, which provides facilities for input and output in human-readable form. Each file is read or written sequentially, as a sequence of characters grouped into lines, and as a sequence of lines grouped into pages. The specification of the package is given below in A.10.1.

2/5 {*AI05-0299-1*} {*AI12-0449-1*} The facilities for file management given above, in A.8.2 and A.8.3, are available for text input-output. In place of Read and Write, however, there are procedures Get and Put that input values of suitable types from text files, and output values to them. These values are provided to the Put procedures, and returned by the Get procedures, in a parameter Item. Several overloaded procedures of these names exist, for different types of Item. These Get procedures analyze the input sequences of characters based on lexical elements (see Clause 2) and return the corresponding values; the Put procedures output the given values as appropriate lexical elements. Procedures Get and Put are also available that input and output individual characters treated as character values rather than as lexical elements. Related to character input are procedures to look ahead at the next character without reading it, and to read a character "immediately" without waiting for an end-of-line to signal availability.

3 In addition to the procedures Get and Put for numeric and enumeration types of Item that operate on text files, analogous procedures are provided that read from and write to a parameter of type String. These procedures perform the same analysis and composition of character sequences as their counterparts which have a file parameter.

4 For all Get and Put procedures that operate on text files, and for many other subprograms, there are forms with and without a file parameter. Each such Get procedure operates on an input file, and each such Put procedure operates on an output file. If no file is specified, a default input file or a default output file is used.

5 At the beginning of program execution the default input and output files are the so-called standard input file and standard output file. These files are open, have respectively the current modes In_File and

Out_File, and are associated with two implementation-defined external files. Procedures are provided to change the current default input file and the current default output file.

Implementation defined: The external files associated with the standard input, standard output, and standard error files. 5.a/2

Implementation Note: {*8652/0113*} {*AI95-00087-01*} The default input file and default output file are not the names of distinct file objects, but rather the *role* played by one or more (other) file object(s). Thus, they generally will be implemented as accesses to another file object. An implementation that implements them by copying them is incorrect. 5.a.1/1

At the beginning of program execution a default file for program-dependent error-related text output is the so-called standard error file. This file is open, has the current mode Out_File, and is associated with an implementation-defined external file. A procedure is provided to change the current default error file. 6

From a logical point of view, a text file is a sequence of pages, a page is a sequence of lines, and a line is a sequence of characters; the end of a line is marked by a *line terminator*; the end of a page is marked by the combination of a line terminator immediately followed by a *page terminator*; and the end of a file is marked by the combination of a line terminator immediately followed by a page terminator and then a *file terminator*. Terminators are generated during output; either by calls of procedures provided expressly for that purpose; or implicitly as part of other operations, for example, when a bounded line length, a bounded page length, or both, have been specified for a file. 7

{*AI12-0444-1*} The actual nature of terminators is not defined by the language and hence depends on the implementation. Although terminators are recognized or generated by certain of the procedures that follow, they are not necessarily implemented as characters or as sequences of characters. Whether they are characters (and if so which ones) in any particular implementation is not of concern to a user who neither explicitly outputs nor explicitly inputs control characters. The effect of input (Get) or output (Put) of control characters (other than horizontal tabulation) is not specified by the language. 8/5

The characters of a line are numbered, starting from one; the number of a character is called its *column number*. For a line terminator, a column number is also defined: it is one more than the number of characters in the line. The lines of a page, and the pages of a file, are similarly numbered. The current column number is the column number of the next character or line terminator to be transferred. The current line number is the number of the current line. The current page number is the number of the current page. These numbers are values of the subtype Positive_Count of the type Count (by convention, the value zero of the type Count is used to indicate special conditions). 9

```
type Count is range 0 .. implementation-defined;
subtype Positive_Count is Count range 1 .. Count'Last;
```
10

For an output file or an append file, a *maximum line length* can be specified and a *maximum page length* can be specified. If a value to be output cannot fit on the current line, for a specified maximum line length, then a new line is automatically started before the value is output; if, further, this new line cannot fit on the current page, for a specified maximum page length, then a new page is automatically started before the value is output. Functions are provided to determine the maximum line length and the maximum page length. When a file is opened with mode Out_File or Append_File, both values are zero: by convention, this means that the line lengths and page lengths are unbounded. (Consequently, output consists of a single line if the subprograms for explicit control of line and page structure are not used.) The constant Unbounded is provided for this purpose. 11

Extensions to Ada 83

Append_File is new in Ada 95. 11.a

A.10.1 The Package Text_IO

Static Semantics

1 The library package Text_IO has the following declaration:

2/5
```
{AI12-0302-1} with Ada.IO_Exceptions;
package Ada.Text_IO
   with Global => in out synchronized is
```

3
```
   type File_Type is limited private;
```

4
```
   type File_Mode is (In_File, Out_File, Append_File);
```

5
```
   type Count is range 0 .. implementation-defined;
   subtype Positive_Count is Count range 1 .. Count'Last;
   Unbounded : constant Count := 0; -- line and page length
```

6
```
   subtype Field       is Integer range 0 .. implementation-defined;
   subtype Number_Base is Integer range 2 .. 16;
```

7
```
   type Type_Set is (Lower_Case, Upper_Case);
```

8
```
   -- File Management
```

9
```
   procedure Create (File : in out File_Type;
                     Mode : in File_Mode := Out_File;
                     Name : in String    := "";
                     Form : in String    := "");
```

10
```
   procedure Open   (File : in out File_Type;
                     Mode : in File_Mode;
                     Name : in String;
                     Form : in String := "");
```

11
```
   procedure Close  (File : in out File_Type);
   procedure Delete (File : in out File_Type);
   procedure Reset  (File : in out File_Type; Mode : in File_Mode);
   procedure Reset  (File : in out File_Type);
```

12
```
   function  Mode   (File : in File_Type) return File_Mode;
   function  Name   (File : in File_Type) return String;
   function  Form   (File : in File_Type) return String;
```

13
```
   function  Is_Open(File : in File_Type) return Boolean;
```

14
```
   -- Control of default input and output files
```

15
```
   procedure Set_Input (File : in File_Type);
   procedure Set_Output(File : in File_Type);
   procedure Set_Error (File : in File_Type);
```

16
```
   function Standard_Input  return File_Type;
   function Standard_Output return File_Type;
   function Standard_Error  return File_Type;
```

17
```
   function Current_Input   return File_Type;
   function Current_Output  return File_Type;
   function Current_Error   return File_Type;
```

18
```
   type File_Access is access constant File_Type;
```

19
```
   function Standard_Input  return File_Access;
   function Standard_Output return File_Access;
   function Standard_Error  return File_Access;
```

20
```
   function Current_Input   return File_Access;
   function Current_Output  return File_Access;
   function Current_Error   return File_Access;
```

21/5
```
   {8652/0051} {AI95-00057-01} {AI12-0302-1} --Buffer control
   procedure Flush (File : in File_Type)
      with Global => overriding in out File;
   procedure Flush
      with Global => in out all;
```

```
   -- Specification of line and page lengths                                    22
{AI12-0302-1}    procedure Set_Line_Length(File : in File_Type; To : in Count)  23/5
      with Global => overriding in out File;
   procedure Set_Line_Length(To   : in Count)
      with Global => in out all;

{AI12-0302-1}    procedure Set_Page_Length(File : in File_Type; To : in Count)  24/5
      with Global => overriding in out File;
   procedure Set_Page_Length(To   : in Count)
      with Global => in out all;

{AI12-0302-1}    function  Line_Length(File : in File_Type) return Count;       25/5
   function  Line_Length return Count
      with Global => in all;

{AI12-0302-1}    function  Page_Length(File : in File_Type) return Count;       26/5
   function  Page_Length return Count
      with Global => in all;

   -- Column, Line, and Page Control                                            27

{AI12-0302-1}    procedure New_Line   (File    : in File_Type;                  28/5
                       Spacing : in Positive_Count := 1)
      with Global => overriding in out File;
   procedure New_Line   (Spacing : in Positive_Count := 1)
      with Global => in out all;

{AI12-0302-1}    procedure Skip_Line  (File    : in File_Type;                  29/5
                       Spacing : in Positive_Count := 1)
      with Global => overriding in out File;
   procedure Skip_Line  (Spacing : in Positive_Count := 1)
      with Global => in out all;

{AI12-0302-1}    function  End_Of_Line(File : in File_Type) return Boolean;     30/5
   function  End_Of_Line return Boolean;

{AI12-0302-1}    procedure New_Page   (File : in File_Type)                      31/5
      with Global => overriding in out File;
   procedure New_Page
      with Global => in out all;

{AI12-0302-1}    procedure Skip_Page  (File : in File_Type)                      32/5
      with Global => overriding in out File;
   procedure Skip_Page
      with Global => in out all;

{AI12-0302-1}    function  End_Of_Page(File : in File_Type) return Boolean;     33/5
   function  End_Of_Page return Boolean
      with Global => in all;

{AI12-0302-1}    function  End_Of_File(File : in File_Type) return Boolean;     34/5
   function  End_Of_File return Boolean
      with Global => in all;

{AI12-0302-1}    procedure Set_Col (File : in File_Type; To : in Positive_Count) 35/5
      with Global => overriding in out File;
   procedure Set_Col (To   : in Positive_Count)
      with Global => in out all;

{AI12-0302-1}    procedure Set_Line(File : in File_Type; To : in Positive_Count) 36/5
      with Global => overriding in out File;
   procedure Set_Line(To   : in Positive_Count)
      with Global => in out all;

{AI12-0302-1}    function Col (File : in File_Type) return Positive_Count;      37/5
   function Col  return Positive_Count
      with Global => in all;

{AI12-0302-1}    function Line(File : in File_Type) return Positive_Count;      38/5
   function Line return Positive_Count
      with Global => in all;

{AI12-0302-1}    function Page(File : in File_Type) return Positive_Count;      39/5
   function Page return Positive_Count
      with Global => in all;
```

```
40          -- Character Input-Output
41/5    {AI12-0302-1}    procedure Get(File : in  File_Type; Item : out Character)
              with Global => overriding in out File;
           procedure Get(Item : out Character)
              with Global => in out all;
42/5    {AI12-0302-1}    procedure Put(File : in  File_Type; Item : in Character)
              with Global => overriding in out File;
           procedure Put(Item : in  Character)
              with Global => in out all;
43/5    {AI12-0302-1}    procedure Look_Ahead (File        : in  File_Type;
                                 Item        : out Character;
                                 End_Of_Line : out Boolean)
              with Global => overriding in out File;
           procedure Look_Ahead (Item        : out Character;
                                 End_Of_Line : out Boolean)
              with Global => in out all;
44/5    {AI12-0302-1}    procedure Get_Immediate(File      : in  File_Type;
                                 Item      : out Character)
              with Global => overriding in out File;
           procedure Get_Immediate(Item      : out Character)
              with Global => in out all;
45/5    {AI12-0302-1}    procedure Get_Immediate(File      : in  File_Type;
                                 Item      : out Character;
                                 Available : out Boolean)
              with Global => overriding in out File;
           procedure Get_Immediate(Item      : out Character;
                                 Available : out Boolean)
              with Global => in out all;
46          -- String Input-Output
47/5    {AI12-0302-1}    procedure Get(File : in  File_Type; Item : out String)
              with Global => overriding in out File;
           procedure Get(Item : out String)
              with Global => in out all;
48/5    {AI12-0302-1}    procedure Put(File : in  File_Type; Item : in String)
              with Global => overriding in out File;
           procedure Put(Item : in  String)
              with Global => in out all;
49/5    {AI12-0302-1}    procedure Get_Line(File : in  File_Type;
                              Item : out String;
                              Last : out Natural)
              with Global => overriding in out File;
           procedure Get_Line(Item : out String; Last : out Natural)
              with Global => in out all;
49.1/5  {AI95-00301-01} {AI12-0302-1}    function Get_Line(File : in  File_Type) return
        String
              with Global => overriding in out File;
           function Get_Line return String
              with Global => in out all;
50/5    {AI12-0302-1}    procedure Put_Line(File : in  File_Type; Item : in String)
              with Global => overriding in out File;
           procedure Put_Line(Item : in  String)
              with Global => in out all;
51          -- Generic packages for Input-Output of Integer Types
52          generic
              type Num is range <>;
           package Integer_IO is
53             Default_Width : Field := Num'Width;
               Default_Base  : Number_Base := 10;
```

```
{AI12-0302-1}       procedure Get(File  : in  File_Type;                    54/5
                                  Item  : out Num;
                                  Width : in Field := 0)
                       with Global => overriding in out File;
                    procedure Get(Item  : out Num;
                                  Width : in  Field := 0)
                       with Global => in out all;
{AI12-0241-1} {AI12-0302-1}       procedure Put(File  : in File_Type;       55/5
                                  Item  : in Num;
                                  Width : in Field := Default_Width;
                                  Base  : in Number_Base := Default_Base)
                       with Global => overriding in out File;
                    procedure Put(Item  : in Num;
                                  Width : in Field := Default_Width;
                                  Base  : in Number_Base := Default_Base)
                       with Global => in out all;
                    procedure Get(From : in  String;
                                  Item : out Num;
                                  Last : out Positive)
                       with Nonblocking;
                    procedure Put(To   : out String;
                                  Item : in Num;
                                  Base : in Number_Base := Default_Base)
                       with Nonblocking;
   end Integer_IO;                                                          56
   generic                                                                  57
      type Num is mod <>;
   package Modular_IO is
      Default_Width : Field := Num'Width;                                   58
      Default_Base  : Number_Base := 10;
{AI12-0302-1}       procedure Get(File  : in  File_Type;                    59/5
                                  Item  : out Num;
                                  Width : in Field := 0)
                       with Global => overriding in out File;
                    procedure Get(Item  : out Num;
                                  Width : in  Field := 0)
                       with Global => in out all;
{AI12-0241-1} {AI12-0302-1}       procedure Put(File  : in File_Type;       60/5
                                  Item  : in Num;
                                  Width : in Field := Default_Width;
                                  Base  : in Number_Base := Default_Base)
                       with Global => overriding in out File;
                    procedure Put(Item  : in Num;
                                  Width : in Field := Default_Width;
                                  Base  : in Number_Base := Default_Base)
                       with Global => in out all;
                    procedure Get(From : in  String;
                                  Item : out Num;
                                  Last : out Positive)
                       with Nonblocking;
                    procedure Put(To   : out String;
                                  Item : in Num;
                                  Base : in Number_Base := Default_Base)
                       with Nonblocking;
   end Modular_IO;                                                          61
   -- Generic packages for Input-Output of Real Types                       62
   generic                                                                  63
      type Num is digits <>;
   package Float_IO is
      Default_Fore : Field := 2;                                            64
      Default_Aft  : Field := Num'Digits-1;
      Default_Exp  : Field := 3;
```

```
65/5    {AI12-0302-1}        procedure Get(File  : in  File_Type;
                             Item  : out Num;
                             Width : in  Field := 0)
                   with Global => overriding in out File;
                procedure Get(Item  : out Num;
                             Width : in  Field := 0)
                   with Global => in out all;
66/5    {AI12-0302-1}        procedure Put(File : in File_Type;
                             Item : in Num;
                             Fore : in Field := Default_Fore;
                             Aft  : in Field := Default_Aft;
                             Exp  : in Field := Default_Exp)
                   with Global => overriding in out File;
                procedure Put(Item : in Num;
                             Fore : in Field := Default_Fore;
                             Aft  : in Field := Default_Aft;
                             Exp  : in Field := Default_Exp)
                   with Global => in out all;
67/5    {AI12-0241-1}        procedure Get(From : in String;
                             Item : out Num;
                             Last : out Positive)
                   with Nonblocking;
                procedure Put(To   : out String;
                             Item : in Num;
                             Aft  : in Field := Default_Aft;
                             Exp  : in Field := Default_Exp)
                   with Nonblocking;
             end Float_IO;
68           generic
                type Num is delta <>;
             package Fixed_IO is
69              Default_Fore : Field := Num'Fore;
                Default_Aft  : Field := Num'Aft;
                Default_Exp  : Field := 0;
70/5    {AI12-0302-1}        procedure Get(File  : in  File_Type;
                             Item  : out Num;
                             Width : in  Field := 0)
                   with Global => overriding in out File;
                procedure Get(Item  : out Num;
                             Width : in  Field := 0)
                   with Global => in out all;
71/5    {AI12-0302-1}        procedure Put(File : in File_Type;
                             Item : in Num;
                             Fore : in Field := Default_Fore;
                             Aft  : in Field := Default_Aft;
                             Exp  : in Field := Default_Exp)
                   with Global => overriding in out File;
                procedure Put(Item : in Num;
                             Fore : in Field := Default_Fore;
                             Aft  : in Field := Default_Aft;
                             Exp  : in Field := Default_Exp)
                   with Global => in out all;
72/5    {AI12-0241-1}        procedure Get(From : in  String;
                             Item : out Num;
                             Last : out Positive)
                   with Nonblocking;
                procedure Put(To   : out String;
                             Item : in Num;
                             Aft  : in Field := Default_Aft;
                             Exp  : in Field := Default_Exp)
                   with Nonblocking;
             end Fixed_IO;
```

```
   generic                                                                    73
      type Num is delta <> digits <>;
   package Decimal_IO is

      Default_Fore : Field := Num'Fore;                                       74
      Default_Aft  : Field := Num'Aft;
      Default_Exp  : Field := 0;

{AI12-0302-1}       procedure Get(File  : in  File_Type;                     75/5
                    Item  : out Num;
                    Width : in  Field := 0)
         with Global => overriding in out File;
      procedure Get(Item  : out Num;
                    Width : in  Field := 0)
         with Global => in out all;

{AI12-0302-1}       procedure Put(File : in File_Type;                       76/5
                    Item : in Num;
                    Fore : in Field := Default_Fore;
                    Aft  : in Field := Default_Aft;
                    Exp  : in Field := Default_Exp)
         with Global => overriding in out File;
      procedure Put(Item : in Num;
                    Fore : in Field := Default_Fore;
                    Aft  : in Field := Default_Aft;
                    Exp  : in Field := Default_Exp)
         with Global => in out all;

{AI12-0241-1}       procedure Get(From : in  String;                         77/5
                    Item : out Num;
                    Last : out Positive)
         with Nonblocking;
      procedure Put(To   : out String;
                    Item : in Num;
                    Aft  : in Field := Default_Aft;
                    Exp  : in Field := Default_Exp)
         with Nonblocking;
   end Decimal_IO;

   -- Generic package for Input-Output of Enumeration Types                   78

   generic                                                                    79
      type Enum is (<>);
   package Enumeration_IO is

      Default_Width   : Field := 0;                                          80
      Default_Setting : Type_Set := Upper_Case;

{AI12-0302-1}       procedure Get(File : in  File_Type;                      81/5
                    Item : out Enum)
         with Global => overriding in out File;
      procedure Get(Item : out Enum)
         with Global => in out all;

{AI12-0302-1}       procedure Put(File  : in File_Type;                      82/5
                    Item  : in Enum;
                    Width : in Field    := Default_Width;
                    Set   : in Type_Set := Default_Setting)
         with Global => overriding in out File;
      procedure Put(Item  : in Enum;
                    Width : in Field    := Default_Width;
                    Set   : in Type_Set := Default_Setting)
         with Global => in out all;

{AI12-0241-1}       procedure Get(From : in  String;                         83/5
                    Item : out Enum;
                    Last : out Positive)
         with Nonblocking;
      procedure Put(To   : out String;
                    Item : in  Enum;
                    Set  : in  Type_Set := Default_Setting)
         with Nonblocking;
   end Enumeration_IO;
```

```
   -- Exceptions
   Status_Error : exception renames IO_Exceptions.Status_Error;
      Mode_Error   : exception renames IO_Exceptions.Mode_Error;
      Name_Error   : exception renames IO_Exceptions.Name_Error;
      Use_Error    : exception renames IO_Exceptions.Use_Error;
      Device_Error : exception renames IO_Exceptions.Device_Error;
      End_Error    : exception renames IO_Exceptions.End_Error;
      Data_Error   : exception renames IO_Exceptions.Data_Error;
      Layout_Error : exception renames IO_Exceptions.Layout_Error;
```

{*AI12-0021-1*}

```
   package Wide_File_Names is
      -- File management
      procedure Create (File : in out File_Type;
                        Mode : in File_Mode := Out_File;
                        Name : in Wide_String := "";
                        Form : in Wide_String := "");
      procedure Open   (File : in out File_Type;
                        Mode : in File_Mode;
                        Name : in Wide_String;
                        Form : in Wide_String := "");
      function Name    (File : in File_Type) return Wide_String;
      function Form    (File : in File_Type) return Wide_String;
   end Wide_File_Names;
```

{*AI12-0021-1*}

```
   package Wide_Wide_File_Names is
      -- File management
      procedure Create (File : in out File_Type;
                        Mode : in File_Mode := Out_File;
                        Name : in Wide_Wide_String := "";
                        Form : in Wide_Wide_String := "");
      procedure Open   (File : in out File_Type;
                        Mode : in File_Mode;
                        Name : in Wide_Wide_String;
                        Form : in Wide_Wide_String := "");
      function Name    (File : in File_Type) return Wide_Wide_String;
      function Form    (File : in File_Type) return Wide_Wide_String;
   end Wide_Wide_File_Names;
private
   ... -- not specified by the language
end Ada.Text_IO;
```

{*AI95-00360-01*} The type File_Type needs finalization (see 7.6).

Incompatibilities With Ada 83

Append_File is a new element of enumeration type File_Mode.

Extensions to Ada 83

Get_Immediate, Look_Ahead, the subprograms for dealing with standard error, the type File_Access and its associated subprograms, and the generic packages Modular_IO and Decimal_IO are new in Ada 95.

Incompatibilities With Ada 95

{*AI95-00360-01*} **Amendment Correction:** Text_IO.File_Type is defined to need finalization. If the restriction No_Nested_Finalization (see D.7) applies to the partition, and File_Type does not have a controlled part, it will not be allowed in local objects in Ada 2005 whereas it would be allowed in original Ada 95. Such code is not portable, as another Ada compiler may have a controlled part in File_Type, and thus would be illegal.

Wording Changes from Ada 95

{*8652/0051*} {*AI95-00057-01*} **Corrigendum:** Corrected the parameter mode of Flush; otherwise it could not be used on Standard_Output. 86.d/2

{*AI95-00301-01*} The Text_IO.Get_Line functions are new; they are described in A.10.7, “Input-Output of Characters and Strings”. 86.e/2

Incompatibilities With Ada 2012

{*AI12-0021-1*} The Wide_File_Names and Wide_Wide_File_Names nested packages are newly added to Ada.Text_IO. Therefore, a use clause conflict is possible; see the introduction of Annex A for more on this topic. 86.f/5

A.10.2 Text File Management

Static Semantics

{*AI12-0449-1*} The only allowed file modes for text files are the modes In_File, Out_File, and Append_File. The subprograms given in A.8.2 for the control of external files, and the function End_Of_File given in A.8.3 for sequential input-output, are also available for text files. There is also a version of End_Of_File that refers to the current default input file. For text files, the procedures have the following additional effects: 1/5

- For the procedures Create and Open: After a file with mode Out_File or Append_File is opened, the page length and line length are unbounded (both have the conventional value zero). After a file (of any mode) is opened, the current column, current line, and current page numbers are set to one. If the mode is Append_File, it is implementation defined whether a page terminator will separate preexisting text in the file from the new text to be written. 2

  Reason: For a file with mode Append_File, although it may seem more sensible for Open to set the current column, line, and page number based on the number of pages in the file, the number of lines on the last page, and the number of columns in the last line, we rejected this approach because of implementation costs; it would require the implementation to scan the file before doing the append, or to do processing that would be equivalent in effect. 2.a

  For similar reasons, there is no requirement to erase the last page terminator of the file, nor to insert an explicit page terminator in the case when the final page terminator of a file is represented implicitly by the implementation. 2.b

- For the procedure Close: If the file has the current mode Out_File or Append_File, has the effect of calling New_Page, unless the current page is already terminated; then outputs a file terminator. 3

- For the procedure Reset: If the file has the current mode Out_File or Append_File, has the effect of calling New_Page, unless the current page is already terminated; then outputs a file terminator. The current column, line, and page numbers are set to one, and the line and page lengths to Unbounded. If the new mode is Append_File, it is implementation defined whether a page terminator will separate preexisting text in the file from the new text to be written. 4

  Reason: The behavior of Reset should be similar to closing a file and reopening it with the given mode 4.a

The exception Mode_Error is propagated by the procedure Reset upon an attempt to change the mode of a file that is the current default input file, the current default output file, or the current default error file. 5

NOTE An implementation can define the Form parameter of Create and Open to control effects including the following: 6

- the interpretation of line and column numbers for an interactive file, and 7
- the interpretation of text formats in a file created by a foreign program. 8

A.10.3 Default Input, Output, and Error Files

Static Semantics

The following subprograms provide for the control of the particular default files that are used when a file parameter is omitted from a Get, Put, or other operation of text input-output described below, or when application-dependent error-related text is to be output. 1

2 `procedure Set_Input(File : in File_Type);`

3 Operates on a file of mode In_File. Sets the current default input file to File.

4 The exception Status_Error is propagated if the given file is not open. The exception Mode_Error is propagated if the mode of the given file is not In_File.

5 `procedure Set_Output(File : in File_Type);`
`procedure Set_Error (File : in File_Type);`

6 Each operates on a file of mode Out_File or Append_File. Set_Output sets the current default output file to File. Set_Error sets the current default error file to File. The exception Status_Error is propagated if the given file is not open. The exception Mode_Error is propagated if the mode of the given file is not Out_File or Append_File.

7 `function Standard_Input return File_Type;`
`function Standard_Input return File_Access;`

8 Returns the standard input file (see A.10), or an access value designating the standard input file, respectively.

9 `function Standard_Output return File_Type;`
`function Standard_Output return File_Access;`

10 Returns the standard output file (see A.10) or an access value designating the standard output file, respectively.

11 `function Standard_Error return File_Type;`
`function Standard_Error return File_Access;`

12/1 {*8652/0052*} {*AI95-00194-01*} Returns the standard error file (see A.10), or an access value designating the standard error file, respectively.

13 The Form strings implicitly associated with the opening of Standard_Input, Standard_Output, and Standard_Error at the start of program execution are implementation defined.

14 `function Current_Input return File_Type;`
`function Current_Input return File_Access;`

15 Returns the current default input file, or an access value designating the current default input file, respectively.

16 `function Current_Output return File_Type;`
`function Current_Output return File_Access;`

17 Returns the current default output file, or an access value designating the current default output file, respectively.

18 `function Current_Error return File_Type;`
`function Current_Error return File_Access;`

19 Returns the current default error file, or an access value designating the current default error file, respectively.

20/1 {*8652/0051*} {*AI95-00057-01*} `procedure Flush (File : in File_Type);`
`procedure Flush;`

21/4 {*AI12-0130-1*} [The effect of Flush is the same as the corresponding subprogram in Sequential_IO (see A.8.2).] If File is not explicitly specified, Current_Output is used.

21.a/4 **Discussion:** {*AI12-0052-1*} For the purpose of determining whether concurrent calls on text input-output subprograms are required to perform as specified, subprograms that implicitly operate on one of the default input-output files are considered to have a parameter of Current_Input or Current_Output (as appropriate). The result of Current_Output is considered to be overlapping with the file given to the latest call of Set_Output (or Standard_Output if Set_Output

hasn't been called); a corresponding consideration applies to the result of Current_Input. See the introduction of Annex A for details.

Erroneous Execution

{*8652/0053*} {*AI95-00063-01*} The execution of a program is erroneous if it invokes an operation on a current default input, default output, or default error file, and if the corresponding file object is closed or no longer exists. 22/1

Ramification: {*8652/0053*} {*AI95-00063-01*} Closing a default file, then setting the default file to another open file before accessing it is not erroneous. 22.a.1/1

This paragraph was deleted.{*8652/0053*} {*AI95-00063-01*} 23/1

NOTE 1 The standard input, standard output, and standard error files cannot be opened, closed, reset, or deleted, because the parameter File of the corresponding procedures has the mode **in out**. 24

NOTE 2 The standard input, standard output, and standard error files are different file objects, but not necessarily different external files. 25

Wording Changes from Ada 95

{*8652/0051*} {*AI95-00057-01*} **Corrigendum:** Corrected the parameter mode of Flush; otherwise it could not be used on Standard_Output. 25.a/2

{*8652/0052*} {*AI95-00194-01*} **Corrigendum:** Corrected Standard_Error so it refers to the correct file. 25.b/2

{*8652/0053*} {*AI95-00063-01*} **Corrigendum:** Clarified that execution is erroneous only when a closed default file is accessed. 25.c/2

Wording Changes from Ada 2012

{*AI12-0130-1*} **Corrigendum:** Moved the definition of Flush to A.8.2, as all input-output packages now have it. 25.d/4

A.10.4 Specification of Line and Page Lengths

Static Semantics

{*AI12-0445-1*} The subprograms described in this subclause are concerned with the line and page structure of a file of mode Out_File or Append_File. They operate either on the file given as the first parameter, or, in the absence of such a file parameter, on the current default output file. They provide for output of text with a specified maximum line length or page length. In these cases, line and page terminators are output implicitly and automatically when necessary. When line and page lengths are unbounded (that is, when they have the conventional value zero), as in the case of a newly opened file, new lines and new pages are only started when explicitly called for. 1/5

In all cases, the exception Status_Error is propagated if the file to be used is not open; the exception Mode_Error is propagated if the mode of the file is not Out_File or Append_File. 2

```
procedure Set_Line_Length(File : in File_Type; To : in Count);
procedure Set_Line_Length(To   : in Count);
```
3

Sets the maximum line length of the specified output or append file to the number of characters specified by To. The value zero for To specifies an unbounded line length. 4

Ramification: The setting does not affect the lengths of lines in the existing file, rather it only influences subsequent output operations. 4.a

The exception Use_Error is propagated if the specified line length is inappropriate for the associated external file. 5

6
```
procedure Set_Page_Length(File : in File_Type; To : in Count);
procedure Set_Page_Length(To   : in Count);
```

7 Sets the maximum page length of the specified output or append file to the number of lines specified by To. The value zero for To specifies an unbounded page length.

8 The exception Use_Error is propagated if the specified page length is inappropriate for the associated external file.

9
```
function Line_Length(File : in File_Type) return Count;
function Line_Length return Count;
```

10 Returns the maximum line length currently set for the specified output or append file, or zero if the line length is unbounded.

11
```
function Page_Length(File : in File_Type) return Count;
function Page_Length return Count;
```

12 Returns the maximum page length currently set for the specified output or append file, or zero if the page length is unbounded.

A.10.5 Operations on Columns, Lines, and Pages

Static Semantics

1 The subprograms described in this subclause provide for explicit control of line and page structure; they operate either on the file given as the first parameter, or, in the absence of such a file parameter, on the appropriate (input or output) current default file. The exception Status_Error is propagated by any of these subprograms if the file to be used is not open.

2
```
procedure New_Line(File : in File_Type; Spacing : in Positive_Count := 1);
procedure New_Line(Spacing : in Positive_Count := 1);
```

3 Operates on a file of mode Out_File or Append_File.

4 For a Spacing of one: Outputs a line terminator and sets the current column number to one. Then increments the current line number by one, except in the case that the current line number is already greater than or equal to the maximum page length, for a bounded page length; in that case a page terminator is output, the current page number is incremented by one, and the current line number is set to one.

5 For a Spacing greater than one, the above actions are performed Spacing times.

6 The exception Mode_Error is propagated if the mode is not Out_File or Append_File.

7
```
procedure Skip_Line(File  : in File_Type; Spacing : in Positive_Count := 1);
procedure Skip_Line(Spacing : in Positive_Count := 1);
```

8 Operates on a file of mode In_File.

9 For a Spacing of one: Reads and discards all characters until a line terminator has been read, and then sets the current column number to one. If the line terminator is not immediately followed by a page terminator, the current line number is incremented by one. Otherwise, if the line terminator is immediately followed by a page terminator, then the page terminator is skipped, the current page number is incremented by one, and the current line number is set to one.

10 For a Spacing greater than one, the above actions are performed Spacing times.

11 The exception Mode_Error is propagated if the mode is not In_File. The exception End_Error is propagated if an attempt is made to read a file terminator.

```
function End_Of_Line(File : in File_Type) return Boolean;
function End_Of_Line return Boolean;
```

{*AI05-0264-1*} Operates on a file of mode In_File. Returns True if a line terminator or a file terminator is next; otherwise, returns False. 13/3

The exception Mode_Error is propagated if the mode is not In_File. 14

```
procedure New_Page(File : in File_Type);
procedure New_Page;
```

Operates on a file of mode Out_File or Append_File. Outputs a line terminator if the current line is not terminated, or if the current page is empty (that is, if the current column and line numbers are both equal to one). Then outputs a page terminator, which terminates the current page. Adds one to the current page number and sets the current column and line numbers to one. 16

The exception Mode_Error is propagated if the mode is not Out_File or Append_File. 17

```
procedure Skip_Page(File : in File_Type);
procedure Skip_Page;
```

Operates on a file of mode In_File. Reads and discards all characters and line terminators until a page terminator has been read. Then adds one to the current page number, and sets the current column and line numbers to one. 19

The exception Mode_Error is propagated if the mode is not In_File. The exception End_Error is propagated if an attempt is made to read a file terminator. 20

```
function End_Of_Page(File : in File_Type) return Boolean;
function End_Of_Page return Boolean;
```

{*AI05-0264-1*} Operates on a file of mode In_File. Returns True if the combination of a line terminator and a page terminator is next, or if a file terminator is next; otherwise, returns False. 22/3

The exception Mode_Error is propagated if the mode is not In_File. 23

```
function End_Of_File(File : in File_Type) return Boolean;
function End_Of_File return Boolean;
```

{*AI05-0264-1*} Operates on a file of mode In_File. Returns True if a file terminator is next, or if the combination of a line, a page, and a file terminator is next; otherwise, returns False. 25/3

The exception Mode_Error is propagated if the mode is not In_File. 26

The following subprograms provide for the control of the current position of reading or writing in a file. In all cases, the default file is the current output file. 27

```
procedure Set_Col(File : in File_Type; To : in Positive_Count);
procedure Set_Col(To   : in Positive_Count);
```

If the file mode is Out_File or Append_File: 29

- If the value specified by To is greater than the current column number, outputs spaces, adding one to the current column number after each space, until the current column number equals the specified value. If the value specified by To is equal to the current column number, there is no effect. If the value specified by To is less than the current column number, has the effect of calling New_Line (with a spacing of one), then outputs (To – 1) spaces, and sets the current column number to the specified value. 30
- The exception Layout_Error is propagated if the value specified by To exceeds Line_Length when the line length is bounded (that is, when it does not have the conventional value zero). 31

If the file mode is In_File: 32

- Reads (and discards) individual characters, line terminators, and page terminators, until the next character to be read has a column number that equals the value specified by To; there is no effect if the current column number already equals this value. Each transfer of a character or terminator maintains the current column, line, and page numbers in the same way as a Get procedure (see A.10.6). (Short lines will be skipped until a line is reached that has a character at the specified column position.)

- The exception End_Error is propagated if an attempt is made to read a file terminator.

```
procedure Set_Line(File : in File_Type; To : in Positive_Count);
procedure Set_Line(To   : in Positive_Count);
```

If the file mode is Out_File or Append_File:

- {*AI05-0038-1*} If the value specified by To is greater than the current line number, has the effect of repeatedly calling New_Line (with a spacing of one), until the current line number equals the specified value. If the value specified by To is equal to the current line number, there is no effect. If the value specified by To is less than the current line number, has the effect of calling New_Page followed, if To is greater than 1, by a call of New_Line with a spacing equal to (To – 1).

- The exception Layout_Error is propagated if the value specified by To exceeds Page_Length when the page length is bounded (that is, when it does not have the conventional value zero).

If the mode is In_File:

- Has the effect of repeatedly calling Skip_Line (with a spacing of one), until the current line number equals the value specified by To; there is no effect if the current line number already equals this value. (Short pages will be skipped until a page is reached that has a line at the specified line position.)

- The exception End_Error is propagated if an attempt is made to read a file terminator.

```
function Col(File : in File_Type) return Positive_Count;
function Col return Positive_Count;
```

Returns the current column number.

The exception Layout_Error is propagated if this number exceeds Count'Last.

```
function Line(File : in File_Type) return Positive_Count;
function Line return Positive_Count;
```

Returns the current line number.

The exception Layout_Error is propagated if this number exceeds Count'Last.

```
function Page(File : in File_Type) return Positive_Count;
function Page return Positive_Count;
```

Returns the current page number.

The exception Layout_Error is propagated if this number exceeds Count'Last.

The column number, line number, or page number are allowed to exceed Count'Last (as a consequence of the input or output of sufficiently many characters, lines, or pages). These events do not cause any exception to be propagated. However, a call of Col, Line, or Page propagates the exception Layout_Error if the corresponding number exceeds Count'Last.

NOTE {*AI12-0440-1*} A page terminator is always skipped whenever the preceding line terminator is skipped. An implementation can represent the combination of these terminators by a single character, provided that it is properly recognized on input.

Inconsistencies With Ada 2005

{*AI05-0038-1*} **Correction:** Fixed a glitch in Set_Line such that we could have called New_Line(0), which would have to raise Constraint_Error. It's now defined to work. The bug occurred in Ada 95 and Ada 2005. It's very unlikely that any real programs depend on this exception being raised. 52.a/3

A.10.6 Get and Put Procedures

Static Semantics

The procedures Get and Put for items of the type Character, String, numeric types, and enumeration types are described in subsequent subclauses. Features of these procedures that are common to most of these types are described in this subclause. The Get and Put procedures for items of type Character and String deal with individual character values; the Get and Put procedures for numeric and enumeration types treat the items as lexical elements. 1

All procedures Get and Put have forms with a file parameter, written first. Where this parameter is omitted, the appropriate (input or output) current default file is understood to be specified. Each procedure Get operates on a file of mode In_File. Each procedure Put operates on a file of mode Out_File or Append_File. 2

All procedures Get and Put maintain the current column, line, and page numbers of the specified file: the effect of each of these procedures upon these numbers is the result of the effects of individual transfers of characters and of individual output or skipping of terminators. Each transfer of a character adds one to the current column number. Each output of a line terminator sets the current column number to one and adds one to the current line number. Each output of a page terminator sets the current column and line numbers to one and adds one to the current page number. For input, each skipping of a line terminator sets the current column number to one and adds one to the current line number; each skipping of a page terminator sets the current column and line numbers to one and adds one to the current page number. Similar considerations apply to the procedures Get_Line, Put_Line, and Set_Col. 3

Several Get and Put procedures, for numeric and enumeration types, have *format* parameters which specify field lengths; these parameters are of the nonnegative subtype Field of the type Integer. 4

{*AI95-00223-01*} Input-output of enumeration values uses the syntax of the corresponding lexical elements. Any Get procedure for an enumeration type begins by skipping any leading blanks, or line or page terminators. A *blank* is defined as a space or a horizontal tabulation character. Next, characters are input only so long as the sequence input is an initial sequence of an identifier or of a character literal (in particular, input ceases when a line terminator is encountered). The character or line terminator that causes input to cease remains available for subsequent input. 5/2

For a numeric type, the Get procedures have a format parameter called Width. If the value given for this parameter is zero, the Get procedure proceeds in the same manner as for enumeration types, but using the syntax of numeric literals instead of that of enumeration literals. If a nonzero value is given, then exactly Width characters are input, or the characters up to a line terminator, whichever comes first; any skipped leading blanks are included in the count. The syntax used for numeric literals is an extended syntax that allows a leading sign (but no intervening blanks, or line or page terminators) and that also allows (for real types) an integer literal as well as forms that have digits only before the point or only after the point. 6

{*AI12-0445-1*} Any Put procedure, for an item of a numeric or an enumeration type, outputs the value of the item as a numeric literal, identifier, or character literal, as appropriate. This is preceded by leading spaces if required by the format parameters Width or Fore (as described in later subclauses), and then a minus sign for a negative value; for an enumeration type, the spaces follow instead of leading. The format given for a Put procedure is overridden if it is insufficiently wide, by using the minimum necessary width. 7/5

8 Two further cases arise for Put procedures for numeric and enumeration types, if the line length of the specified output file is bounded (that is, if it does not have the conventional value zero). If the number of characters to be output does not exceed the maximum line length, but is such that they cannot fit on the current line, starting from the current column, then (in effect) New_Line is called (with a spacing of one) before output of the item. Otherwise, if the number of characters exceeds the maximum line length, then the exception Layout_Error is propagated and nothing is output.

9 The exception Status_Error is propagated by any of the procedures Get, Get_Line, Put, and Put_Line if the file to be used is not open. The exception Mode_Error is propagated by the procedures Get and Get_Line if the mode of the file to be used is not In_File; and by the procedures Put and Put_Line, if the mode is not Out_File or Append_File.

10 The exception End_Error is propagated by a Get procedure if an attempt is made to skip a file terminator. The exception Data_Error is propagated by a Get procedure if the sequence finally input is not a lexical element corresponding to the type, in particular if no characters were input; for this test, leading blanks are ignored; for an item of a numeric type, when a sign is input, this rule applies to the succeeding numeric literal. The exception Layout_Error is propagated by a Put procedure that outputs to a parameter of type String, if the length of the actual string is insufficient for the output of the item.

Examples

11/5 {*AI12-0449-1*} In the examples, here and in A.10.8 and A.10.9, the string quotes and the lower case letter b are not transferred: they are shown only to reveal the layout and spaces.

12
```
N : Integer;
   ...
Get(N);
```

13
```
--  Characters at input   Sequence input    Value of N

--  bb–12535b             –12535            –12535
--  bb12_535e1b           12_535e1          125350
--  bb12_535e;            12_535e           (none) Data_Error raised
```

14 Example of overridden width parameter:

15
```
Put(Item => -23, Width => 2);  -- "–23"
```

Wording Changes from Ada 95

15.a/2 {*AI95-00223-01*} Removed conflicting text describing the skipping of blanks for a Get procedure.

A.10.7 Input-Output of Characters and Strings

Static Semantics

1 For an item of type Character the following procedures are provided:

2
```
procedure Get(File : in File_Type; Item : out Character);
procedure Get(Item : out Character);
```

3 After skipping any line terminators and any page terminators, reads the next character from the specified input file and returns the value of this character in the out parameter Item.

4 The exception End_Error is propagated if an attempt is made to skip a file terminator.

5
```
procedure Put(File : in File_Type; Item : in Character);
procedure Put(Item : in Character);
```

6 If the line length of the specified output file is bounded (that is, does not have the conventional value zero), and the current column number exceeds it, has the effect of calling New_Line with a spacing of one. Then, or otherwise, outputs the given character to the file.

```
procedure Look_Ahead (File        : in  File_Type;                    7
                      Item        : out Character;
                      End_Of_Line : out Boolean);
procedure Look_Ahead (Item        : out Character;
                      End_Of_Line : out Boolean);
```

{*AI05-0038-1*} {*AI05-0264-1*} Status_Error is propagated if the file is not open. Mode_Error is propagated if the mode of the file is not In_File. Sets End_Of_Line to True if at end of line, including if at end of page or at end of file; in each of these cases the value of Item is not specified. Otherwise, End_Of_Line is set to False and Item is set to the next character (without consuming it) from the file. 8/3

```
procedure Get_Immediate(File : in  File_Type;                         9
                        Item : out Character);
procedure Get_Immediate(Item : out Character);
```

{*AI05-0038-1*} Reads the next character, either control or graphic, from the specified File or the default input file. Status_Error is propagated if the file is not open. Mode_Error is propagated if the mode of the file is not In_File. End_Error is propagated if at the end of the file. The current column, line and page numbers for the file are not affected. 10/3

```
procedure Get_Immediate(File      : in  File_Type;                    11
                        Item      : out Character;
                        Available : out Boolean);
procedure Get_Immediate(Item      : out Character;
                        Available : out Boolean);
```

{*AI05-0038-1*} If a character, either control or graphic, is available from the specified File or the default input file, then the character is read; Available is True and Item contains the value of this character. If a character is not available, then Available is False and the value of Item is not specified. Status_Error is propagated if the file is not open. Mode_Error is propagated if the mode of the file is not In_File. End_Error is propagated if at the end of the file. The current column, line and page numbers for the file are not affected. 12/3

{*AI95-00301-01*} For an item of type String the following subprograms are provided: 13/2

```
procedure Get(File : in File_Type; Item : out String);                14
procedure Get(Item : out String);
```

Determines the length of the given string and attempts that number of Get operations for successive characters of the string (in particular, no operation is performed if the string is null). 15

```
procedure Put(File : in File_Type; Item : in String);                 16
procedure Put(Item : in String);
```

Determines the length of the given string and attempts that number of Put operations for successive characters of the string (in particular, no operation is performed if the string is null). 17

```
function Get_Line(File : in File_Type) return String;                 17.1/2
function Get_Line return String;
```

{*AI95-00301-01*} Returns a result string constructed by reading successive characters from the specified input file, and assigning them to successive characters of the result string. The result string has a lower bound of 1 and an upper bound of the number of characters read. Reading stops when the end of the line is met; Skip_Line is then (in effect) called with a spacing of 1. 17.2/2

{*AI95-00301-01*} Constraint_Error is raised if the length of the line exceeds Positive'Last; in this case, the line number and page number are unchanged, and the column number is unspecified but no less than it was before the call. The exception End_Error is propagated if an attempt is made to skip a file terminator. 17.3/2

17.a/2 **Ramification:** {*AI95-00301-01*} Precisely what is left in the file is unspecified if Constraint_Error is raised because the line doesn't fit in a String; it should be consistent with column number. This allows implementers to use whatever buffering scheme makes sense. But the line terminator is not skipped in this case.

18
```
procedure Get_Line(File : in File_Type;
                   Item : out String;
                   Last : out Natural);
procedure Get_Line(Item : out String;
                   Last : out Natural);
```

19 Reads successive characters from the specified input file and assigns them to successive characters of the specified string. Reading stops if the end of the string is met. Reading also stops if the end of the line is met before meeting the end of the string; in this case Skip_Line is (in effect) called with a spacing of 1. The values of characters not assigned are not specified.

20 If characters are read, returns in Last the index value such that Item(Last) is the last character assigned (the index of the first character assigned is Item'First). If no characters are read, returns in Last an index value that is one less than Item'First. The exception End_Error is propagated if an attempt is made to skip a file terminator.

21
```
procedure Put_Line(File : in File_Type; Item : in String);
procedure Put_Line(Item : in String);
```

22 Calls the procedure Put for the given string, and then the procedure New_Line with a spacing of one.

Implementation Advice

23 The Get_Immediate procedures should be implemented with unbuffered input. For a device such as a keyboard, input should be “available” if a key has already been typed, whereas for a disk file, input should always be available except at end of file. For a file associated with a keyboard-like device, any line-editing features of the underlying operating system should be disabled during the execution of Get_Immediate.

23.a/2 **Implementation Advice:** Get_Immediate should be implemented with unbuffered input; input should be available immediately; line-editing should be disabled.

24 NOTE 1 Get_Immediate can be used to read a single key from the keyboard “immediately”; that is, without waiting for an end of line. In a call of Get_Immediate without the parameter Available, the caller will wait until a character is available.

25 NOTE 2 In a literal string parameter of Put, the enclosing string bracket characters are not output. Each doubled string bracket character in the enclosed string is output as a single string bracket character, as a consequence of the rule for string literals (see 2.6).

26 NOTE 3 A string read by Get or written by Put can extend over several lines. An implementation is allowed to assume that certain external files do not contain page terminators, in which case Get_Line and Skip_Line can return as soon as a line terminator is read.

Incompatibilities With Ada 95

26.a/3 {*AI95-00301-01*} {*AI05-0005-1*} The Get_Line functions are added to Ada.Text_IO. If Ada.Text_IO is referenced in a use_clause, and a function Get_Line is defined in a package that is also referenced in a use_clause, the user-defined Get_Line may no longer be use-visible, resulting in errors. This should be rare and is easily fixed if it does occur.

Extensions to Ada 95

26.b/2 {*AI95-00301-01*} The Text_IO.Get_Line functions are new.

Wording Changes from Ada 2005

26.c/3 {*AI05-0038-1*} **Correction:** Added missing wording about raising Status_Error to Look_Ahead and Get_Immediate.

A.10.8 Input-Output for Integer Types

Static Semantics

The following procedures are defined in the generic packages Integer_IO and Modular_IO, which have to be instantiated for the appropriate signed integer or modular type respectively (indicated by Num in the specifications). 1

Values are output as decimal or based literals, without low line characters or exponent, and, for Integer_IO, preceded by a minus sign if negative. The format (which includes any leading spaces and minus sign) can be specified by an optional field width parameter. Values of widths of fields in output formats are of the nonnegative integer subtype Field. Values of bases are of the integer subtype Number_Base. 2

```
subtype Number_Base is Integer range 2 .. 16;
```
3

The default field width and base to be used by output procedures are defined by the following variables that are declared in the generic packages Integer_IO and Modular_IO: 4

```
Default_Width : Field := Num'Width;
Default_Base  : Number_Base := 10;
```
5

The following procedures are provided: 6

```
procedure Get(File : in File_Type; Item : out Num; Width : in Field := 0);
procedure Get(Item : out Num; Width : in Field := 0);
```
7

If the value of the parameter Width is zero, skips any leading blanks, line terminators, or page terminators, then reads a plus sign if present or (for a signed type only) a minus sign if present, then reads the longest possible sequence of characters matching the syntax of a numeric literal without a point. If a nonzero value of Width is supplied, then exactly Width characters are input, or the characters (possibly none) up to a line terminator, whichever comes first; any skipped leading blanks are included in the count. 8

Returns, in the parameter Item, the value of type Num that corresponds to the sequence input. 9

{*AI05-0038-1*} The exception Data_Error is propagated if the sequence of characters read does not form a legal integer literal or if the value obtained is not of the subtype Num. 10/3

```
procedure Put(File  : in File_Type;
              Item  : in Num;
              Width : in Field := Default_Width;
              Base  : in Number_Base := Default_Base);

procedure Put(Item  : in Num;
              Width : in Field := Default_Width;
              Base  : in Number_Base := Default_Base);
```
11

Outputs the value of the parameter Item as an integer literal, with no low lines, no exponent, and no leading zeros (but a single zero for the value zero), and a preceding minus sign for a negative value. 12

If the resulting sequence of characters to be output has fewer than Width characters, then leading spaces are first output to make up the difference. 13

Uses the syntax for decimal literal if the parameter Base has the value ten (either explicitly or through Default_Base); otherwise, uses the syntax for based literal, with any letters in upper case. 14

15 `procedure Get(From : in String; Item : out Num; Last : out Positive);`

16 Reads an integer value from the beginning of the given string, following the same rules as the Get procedure that reads an integer value from a file, but treating the end of the string as a file terminator. Returns, in the parameter Item, the value of type Num that corresponds to the sequence input. Returns in Last the index value such that From(Last) is the last character read.

17 The exception Data_Error is propagated if the sequence input does not have the required syntax or if the value obtained is not of the subtype Num.

18
```
procedure Put(To   : out String;
              Item : in Num;
              Base : in Number_Base := Default_Base);
```

19 Outputs the value of the parameter Item to the given string, following the same rule as for output to a file, using the length of the given string as the value for Width.

20 Integer_Text_IO is a library package that is a nongeneric equivalent to Text_IO.Integer_IO for the predefined type Integer:

21
```
with Ada.Text_IO;
package Ada.Integer_Text_IO is new Ada.Text_IO.Integer_IO(Integer);
```

22 For each predefined signed integer type, a nongeneric equivalent to Text_IO.Integer_IO is provided, with names such as Ada.Long_Integer_Text_IO.

Implementation Permissions

23/5 {*AI12-0444-1*} The nongeneric equivalent packages can be actual instantiations of the generic package for the appropriate predefined type, though that is not required.

24/3 *This paragraph was deleted.*{*AI05-0038-1*}

Examples

{*AI12-0429-1*} *Examples of use of an instantiation of Text_IO.Integer_IO:*

25/3
```
{AI05-0298-1} subtype Byte_Int is Integer range -127 .. 127;
package Int_IO is new Integer_IO(Byte_Int); use Int_IO;
-- default format used at instantiation,
-- Default_Width = 4, Default_Base = 10
```

26
```
Put(126);                              -- "b126"
Put(-126, 7);                          -- "bbb–126"
Put(126, Width => 13, Base => 2);      -- "bbb2#1111110#"
```

Inconsistencies With Ada 2005

26.a/3 {*AI05-0038-1*} **Correction:** Changed wording to make Integer_IO and Modular_IO raise Data_Error in the same way when the bounds of the subtype are exceeded. There is no value to different behavior, and all surveyed compilers already treat integer and modular values the same way. This could only cause a problem if a program was compiled with some unsurveyed compiler, and the Ada 95-defined behavior is expected for Modular_IO. But note that such code is not portable anyway, as most widely used compilers behave consistently with the new wording, so it is unlikely that such code exists.

A.10.9 Input-Output for Real Types

Static Semantics

1 The following procedures are defined in the generic packages Float_IO, Fixed_IO, and Decimal_IO, which have to be instantiated for the appropriate floating point, ordinary fixed point, or decimal fixed point type respectively (indicated by Num in the specifications).

Values are output as decimal literals without low line characters. The format of each value output consists of a Fore field, a decimal point, an Aft field, and (if a nonzero Exp parameter is supplied) the letter E and an Exp field. The two possible formats thus correspond to: 2

```
Fore  .  Aft
```
3

and to: 4

```
Fore  .  Aft  E  Exp
```
5

without any spaces between these fields. The Fore field may include leading spaces, and a minus sign for negative values. The Aft field includes only decimal digits (possibly with trailing zeros). The Exp field includes the sign (plus or minus) and the exponent (possibly with leading zeros). 6

For floating point types, the default lengths of these fields are defined by the following variables that are declared in the generic package Float_IO: 7

```
Default_Fore : Field := 2;
Default_Aft  : Field := Num'Digits-1;
Default_Exp  : Field := 3;
```
8

For ordinary or decimal fixed point types, the default lengths of these fields are defined by the following variables that are declared in the generic packages Fixed_IO and Decimal_IO, respectively: 9

```
Default_Fore : Field := Num'Fore;
Default_Aft  : Field := Num'Aft;
Default_Exp  : Field := 0;
```
10

The following procedures are provided: 11

```
procedure Get(File : in File_Type; Item : out Num; Width : in Field := 0);
procedure Get(Item : out Num; Width : in Field := 0);
```
12

If the value of the parameter Width is zero, skips any leading blanks, line terminators, or page terminators, then reads the longest possible sequence of characters matching the syntax of any of the following (see 2.4): 13

- [+|–]numeric_literal 14
- [+|–]numeral.[exponent] 15
- [+|–].numeral[exponent] 16
- [+|–]base#based_numeral.#[exponent] 17
- [+|–]base#.based_numeral#[exponent] 18

If a nonzero value of Width is supplied, then exactly Width characters are input, or the characters (possibly none) up to a line terminator, whichever comes first; any skipped leading blanks are included in the count. 19

Returns in the parameter Item the value of type Num that corresponds to the sequence input, preserving the sign (positive if none has been specified) of a zero value if Num is a floating point type and Num'Signed_Zeros is True. 20

The exception Data_Error is propagated if the sequence input does not have the required syntax or if the value obtained is not of the subtype Num. 21

22
```
procedure Put(File : in File_Type;
              Item : in Num;
              Fore : in Field := Default_Fore;
              Aft  : in Field := Default_Aft;
              Exp  : in Field := Default_Exp);

procedure Put(Item : in Num;
              Fore : in Field := Default_Fore;
              Aft  : in Field := Default_Aft;
              Exp  : in Field := Default_Exp);
```

23 Outputs the value of the parameter Item as a decimal literal with the format defined by Fore, Aft and Exp. If the value is negative, or if Num is a floating point type where Num'Signed_Zeros is True and the value is a negatively signed zero, then a minus sign is included in the integer part. If Exp has the value zero, then the integer part to be output has as many digits as are needed to represent the integer part of the value of Item, overriding Fore if necessary, or consists of the digit zero if the value of Item has no integer part.

24 If Exp has a value greater than zero, then the integer part to be output has a single digit, which is nonzero except for the value 0.0 of Item.

25 In both cases, however, if the integer part to be output has fewer than Fore characters, including any minus sign, then leading spaces are first output to make up the difference. The number of digits of the fractional part is given by Aft, or is one if Aft equals zero. The value is rounded; a value of exactly one half in the last place is rounded away from zero.

26 If Exp has the value zero, there is no exponent part. If Exp has a value greater than zero, then the exponent part to be output has as many digits as are needed to represent the exponent part of the value of Item (for which a single digit integer part is used), and includes an initial sign (plus or minus). If the exponent part to be output has fewer than Exp characters, including the sign, then leading zeros precede the digits, to make up the difference. For the value 0.0 of Item, the exponent has the value zero.

27
```
procedure Get(From : in String; Item : out Num; Last : out Positive);
```

28 Reads a real value from the beginning of the given string, following the same rule as the Get procedure that reads a real value from a file, but treating the end of the string as a file terminator. Returns, in the parameter Item, the value of type Num that corresponds to the sequence input. Returns in Last the index value such that From(Last) is the last character read.

29 The exception Data_Error is propagated if the sequence input does not have the required syntax, or if the value obtained is not of the subtype Num.

30
```
procedure Put(To   : out String;
              Item : in Num;
              Aft  : in Field := Default_Aft;
              Exp  : in Field := Default_Exp);
```

31 Outputs the value of the parameter Item to the given string, following the same rule as for output to a file, using a value for Fore such that the sequence of characters output exactly fills the string, including any leading spaces.

32 Float_Text_IO is a library package that is a nongeneric equivalent to Text_IO.Float_IO for the predefined type Float:

33
```
with Ada.Text_IO;
package Ada.Float_Text_IO is new Ada.Text_IO.Float_IO(Float);
```

34 For each predefined floating point type, a nongeneric equivalent to Text_IO.Float_IO is provided, with names such as Ada.Long_Float_Text_IO.

Implementation Permissions

An implementation may extend Get [and Put] for floating point types to support special values such as infinities and NaNs. 35

Discussion: {*AI05-0005-1*} See also the similar permission for the Wide_Wide_Value, Wide_Value, and Value attributes in 3.5. 35.a/3

{*AI12-0444-1*} The implementation of Put may produce an output value with no greater accuracy than that which is supported for the base subtype. The additional accuracy, if any, of the value produced by Put when the number of requested digits in the integer and fractional parts exceeds the required accuracy is implementation defined. 36/5

Discussion: The required accuracy is thus Num'Base'Digits digits if Num is a floating point subtype. For a fixed point subtype the required accuracy is a function of the subtype's Fore, Aft, and Delta attributes. 36.a

Implementation defined: The accuracy of the value produced by Put. 36.b

{*AI12-0444-1*} The nongeneric equivalent packages can be actual instantiations of the generic package for the appropriate predefined type, though that is not required. 37/5

NOTE 1 For an item with a positive value, if output to a string exactly fills the string without leading spaces, then output of the corresponding negative value will propagate Layout_Error. 38

NOTE 2 The rules for the Value attribute (see 3.5) and the rules for Get are based on the same set of formats. 39

Examples

{*AI12-0429-1*} *Examples of use of an instantiation of Text_IO.Float_IO:* 40/5

```
package Real_IO is new Float_IO(Real); use Real_IO;                    41
-- default format used at instantiation, Default_Exp = 3

X : Real := -123.4567;  -- digits 8    (see 3.5.7)                     42

Put(X);   -- default format                          "–1.2345670E+02"  43
Put(X, Fore => 5, Aft => 3, Exp => 2);    -- "bbb–1.235E+2"
Put(X, 5, 3, 0);                                     -- "b–123.457"
```

A.10.10 Input-Output for Enumeration Types

Static Semantics

The following procedures are defined in the generic package Enumeration_IO, which has to be instantiated for the appropriate enumeration type (indicated by Enum in the specification). 1

Values are output using either upper or lower case letters for identifiers. This is specified by the parameter Set, which is of the enumeration type Type_Set. 2

```
type Type_Set is (Lower_Case, Upper_Case);                             3
```

The format (which includes any trailing spaces) can be specified by an optional field width parameter. The default field width and letter case are defined by the following variables that are declared in the generic package Enumeration_IO: 4

```
Default_Width   : Field := 0;                                          5
Default_Setting : Type_Set := Upper_Case;
```

The following procedures are provided: 6

```
procedure Get(File : in File_Type; Item : out Enum);                   7
procedure Get(Item : out Enum);
```

After skipping any leading blanks, line terminators, or page terminators, reads an identifier according to the syntax of this lexical element (lower and upper case being considered equivalent), or a character literal according to the syntax of this lexical element (including the 8

apostrophes). Returns, in the parameter Item, the value of type Enum that corresponds to the sequence input.

9 The exception Data_Error is propagated if the sequence input does not have the required syntax, or if the identifier or character literal does not correspond to a value of the subtype Enum.

10
```
procedure Put(File  : in File_Type;
              Item  : in Enum;
              Width : in Field := Default_Width;
              Set   : in Type_Set := Default_Setting);

procedure Put(Item  : in Enum;
              Width : in Field := Default_Width;
              Set   : in Type_Set := Default_Setting);
```

11 Outputs the value of the parameter Item as an enumeration literal (either an identifier or a character literal). The optional parameter Set indicates whether lower case or upper case is used for identifiers; it has no effect for character literals. If the sequence of characters produced has fewer than Width characters, then trailing spaces are finally output to make up the difference. If Enum is a character type, the sequence of characters produced is as for Enum'Image(Item), as modified by the Width and Set parameters.

11.a/3 **Discussion:** {*AI05-0005-1*} For a character type, the literal might be a Wide_Wide_Character, Wide_Character, or a control character. Whatever Image does for these things is appropriate here, too.

11.b/3 {*AI05-0036-1*} The "characters produced" defines the "characters to be output" in the sense of A.10.6, so a result that cannot fit on any bounded line will raise Layout_Error.

12
```
procedure Get(From : in String; Item : out Enum; Last : out Positive);
```

13 Reads an enumeration value from the beginning of the given string, following the same rule as the Get procedure that reads an enumeration value from a file, but treating the end of the string as a file terminator. Returns, in the parameter Item, the value of type Enum that corresponds to the sequence input. Returns in Last the index value such that From(Last) is the last character read.

14 The exception Data_Error is propagated if the sequence input does not have the required syntax, or if the identifier or character literal does not correspond to a value of the subtype Enum.

14.a/3 **To be honest:** {*AI05-0005-1*} For a character type, it is permissible for the implementation to make Get do the inverse of what Put does, in the case of wide and wide_wide character_literals and control characters.

15
```
procedure Put(To   : out String;
              Item : in Enum;
              Set  : in Type_Set := Default_Setting);
```

16 Outputs the value of the parameter Item to the given string, following the same rule as for output to a file, using the length of the given string as the value for Width.

17/1 {*8652/0054*} {*AI95-00007-01*} Although the specification of the generic package Enumeration_IO would allow instantiation for an integer type, this is not the intended purpose of this generic package, and the effect of such instantiations is not defined by the language.

18 NOTE 1 There is a difference between Put defined for characters, and for enumeration values. Thus

19
```
Ada.Text_IO.Put('A');  -- outputs the character A
```

20
```
package Char_IO is new Ada.Text_IO.Enumeration_IO(Character);
Char_IO.Put('A');  -- outputs the character 'A', between apostrophes
```

21 NOTE 2 The type Boolean is an enumeration type, hence Enumeration_IO can be instantiated for this type.

Wording Changes from Ada 95

21.a/2 {*8652/0054*} {*AI95-00007-01*} **Corrigendum:** Corrected the wording to say Enumeration_IO can be instantiated with an integer type, not a float type.

A.10.11 Input-Output for Bounded Strings

{*AI95-00428-01*} The package Text_IO.Bounded_IO provides input-output in human-readable form for Bounded_Strings. 1/2

Static Semantics

{*AI95-00428-01*} The generic library package Text_IO.Bounded_IO has the following declaration: 2/2

```
{AI12-0302-1} with Ada.Strings.Bounded;                                    3/5
generic
   with package Bounded is
                     new Ada.Strings.Bounded.Generic_Bounded_Length (<>);
package Ada.Text_IO.Bounded_IO
   with Global => in out synchronized is

   procedure Put                                                           4/2
      (File : in File_Type;
       Item : in Bounded.Bounded_String);

   procedure Put                                                           5/2
      (Item : in Bounded.Bounded_String);

   procedure Put_Line                                                      6/2
      (File : in File_Type;
       Item : in Bounded.Bounded_String);

   procedure Put_Line                                                      7/2
      (Item : in Bounded.Bounded_String);

   function Get_Line                                                       8/2
      (File : in File_Type)
      return Bounded.Bounded_String;

   function Get_Line                                                       9/2
      return Bounded.Bounded_String;

   procedure Get_Line                                                      10/2
      (File : in File_Type; Item : out Bounded.Bounded_String);

   procedure Get_Line                                                      11/2
      (Item : out Bounded.Bounded_String);

end Ada.Text_IO.Bounded_IO;                                                12/2
```

{*AI95-00428-01*} For an item of type Bounded_String, the following subprograms are provided: 13/2

```
procedure Put                                                              14/2
   (File : in File_Type;
    Item : in Bounded.Bounded_String);
```

{*AI95-00428-01*} Equivalent to Text_IO.Put (File, Bounded.To_String(Item)); 15/2

```
procedure Put                                                              16/2
   (Item : in Bounded.Bounded_String);
```

{*AI95-00428-01*} Equivalent to Text_IO.Put (Bounded.To_String(Item)); 17/2

```
procedure Put_Line                                                         18/2
   (File : in File_Type;
    Item : in Bounded.Bounded_String);
```

{*AI95-00428-01*} Equivalent to Text_IO.Put_Line (File, Bounded.To_String(Item)); 19/2

```
procedure Put_Line                                                         20/2
   (Item : in Bounded.Bounded_String);
```

{*AI95-00428-01*} Equivalent to Text_IO.Put_Line (Bounded.To_String(Item)); 21/2

22/2
```
function Get_Line
   (File : in File_Type)
   return Bounded.Bounded_String;
```

23/2 {*AI95-00428-01*} Returns Bounded.To_Bounded_String(Text_IO.Get_Line(File));

24/2
```
function Get_Line
   return Bounded.Bounded_String;
```

25/2 {*AI95-00428-01*} Returns Bounded.To_Bounded_String(Text_IO.Get_Line);

26/2
```
procedure Get_Line
   (File : in File_Type; Item : out Bounded.Bounded_String);
```

27/2 {*AI95-00428-01*} Equivalent to Item := Get_Line (File);

28/2
```
procedure Get_Line
   (Item : out Bounded.Bounded_String);
```

29/2 {*AI95-00428-01*} Equivalent to Item := Get_Line;

Extensions to Ada 95

29.a/2 {*AI95-00428-01*} Package Text_IO.Bounded_IO is new.

A.10.12 Input-Output for Unbounded Strings

1/2 {*AI95-00301-01*} The package Text_IO.Unbounded_IO provides input-output in human-readable form for Unbounded_Strings.

Static Semantics

2/2 {*AI95-00301-01*} The library package Text_IO.Unbounded_IO has the following declaration:

3/5
```
{AI12-0302-1} with Ada.Strings.Unbounded;
package Ada.Text_IO.Unbounded_IO
   with Global => in out synchronized is
```

4/2
```
   procedure Put
      (File : in File_Type;
       Item : in Strings.Unbounded.Unbounded_String);
```

5/2
```
   procedure Put
      (Item : in Strings.Unbounded.Unbounded_String);
```

6/2
```
   procedure Put_Line
      (File : in File_Type;
       Item : in Strings.Unbounded.Unbounded_String);
```

7/2
```
   procedure Put_Line
      (Item : in Strings.Unbounded.Unbounded_String);
```

8/2
```
   function Get_Line
      (File : in File_Type)
      return Strings.Unbounded.Unbounded_String;
```

9/2
```
   function Get_Line
      return Strings.Unbounded.Unbounded_String;
```

10/2
```
   procedure Get_Line
      (File : in File_Type; Item : out Strings.Unbounded.Unbounded_String);
```

11/2
```
   procedure Get_Line
      (Item : out Strings.Unbounded.Unbounded_String);
```

12/2
```
end Ada.Text_IO.Unbounded_IO;
```

13/2 {*AI95-00301-01*} For an item of type Unbounded_String, the following subprograms are provided:

```
procedure Put                                                                    14/2
   (File : in File_Type;
    Item : in Strings.Unbounded.Unbounded_String);
```

{*AI95-00301-01*} Equivalent to Text_IO.Put (File, Strings.Unbounded.To_String(Item)); 15/2

```
procedure Put                                                                    16/2
   (Item : in Strings.Unbounded.Unbounded_String);
```

{*AI95-00301-01*} Equivalent to Text_IO.Put (Strings.Unbounded.To_String(Item)); 17/2

```
procedure Put_Line                                                               18/2
   (File : in File_Type;
    Item : in Strings.Unbounded.Unbounded_String);
```

{*AI95-00301-01*} Equivalent to Text_IO.Put_Line (File, Strings.Unbounded.To_String(Item)); 19/2

```
procedure Put_Line                                                               20/2
   (Item : in Strings.Unbounded.Unbounded_String);
```

{*AI95-00301-01*} Equivalent to Text_IO.Put_Line (Strings.Unbounded.To_String(Item)); 21/2

```
function Get_Line                                                                22/2
   (File : in File_Type)
   return Strings.Unbounded.Unbounded_String;
```

{*AI95-00301-01*} Returns Strings.Unbounded.To_Unbounded_String(Text_IO.Get_Line(File)); 23/2

```
function Get_Line                                                                24/2
   return Strings.Unbounded.Unbounded_String;
```

{*AI95-00301-01*} Returns Strings.Unbounded.To_Unbounded_String(Text_IO.Get_Line); 25/2

```
procedure Get_Line                                                               26/2
   (File : in File_Type; Item : out Strings.Unbounded.Unbounded_String);
```

{*AI95-00301-01*} Equivalent to Item := Get_Line (File); 27/2

```
procedure Get_Line                                                               28/2
   (Item : out Strings.Unbounded.Unbounded_String);
```

{*AI95-00301-01*} Equivalent to Item := Get_Line; 29/2

Extensions to Ada 95

{*AI95-00301-01*} Package Text_IO.Unbounded_IO is new. 29.a/2

A.11 Wide Text Input-Output and Wide Wide Text Input-Output

{*AI95-00285-01*} The packages Wide_Text_IO and Wide_Wide_Text_IO provide facilities for input and output in human-readable form. Each file is read or written sequentially, as a sequence of wide characters (or wide wide characters) grouped into lines, and as a sequence of lines grouped into pages. 1/2

Static Semantics

{*AI95-00285-01*} {*AI95-00301-01*} The specification of package Wide_Text_IO is the same as that for Text_IO, except that in each Get, Look_Ahead, Get_Immediate, Get_Line, Put, and Put_Line subprogram, any occurrence of Character is replaced by Wide_Character, and any occurrence of String is replaced by Wide_String. Nongeneric equivalents of Wide_Text_IO.Integer_IO and Wide_Text_IO.-Float_IO are provided (as for Text_IO) for each predefined numeric type, with names such as Ada.Integer_Wide_Text_IO, Ada.Long_Integer_Wide_Text_IO, Ada.Float_Wide_Text_IO, Ada.Long_-Float_Wide_Text_IO. 2/2

{*AI95-00285-01*} {*AI95-00301-01*} The specification of package Wide_Wide_Text_IO is the same as that for Text_IO, except that in each Get, Look_Ahead, Get_Immediate, Get_Line, Put, and Put_Line subprogram, any occurrence of Character is replaced by Wide_Wide_Character, and any occurrence of 3/2

String is replaced by Wide_Wide_String. Nongeneric equivalents of Wide_Wide_Text_IO.Integer_IO and Wide_Wide_Text_IO.Float_IO are provided (as for Text_IO) for each predefined numeric type, with names such as Ada.Integer_Wide_Wide_Text_IO, Ada.Long_Integer_Wide_Wide_Text_IO, Ada.Float_-Wide_Wide_Text_IO, Ada.Long_Float_Wide_Wide_Text_IO.

4/3 {*AI95-00285-01*} {*AI95-00428-01*} {*AI05-0004-1*} {*AI05-0092-1*} The specification of package Wide_Text_IO.Wide_Bounded_IO is the same as that for Text_IO.Bounded_IO, except that any occurrence of Bounded_String is replaced by Bounded_Wide_String, and any occurrence of package Bounded is replaced by Wide_Bounded. The specification of package Wide_Wide_-Text_IO.Wide_Wide_Bounded_IO is the same as that for Text_IO.Bounded_IO, except that any occurrence of Bounded_String is replaced by Bounded_Wide_Wide_String, and any occurrence of package Bounded is replaced by Wide_Wide_Bounded.

4.a/3 **To be honest:** {*AI05-0005-1*} “package Bounded” refers to both the package Ada.Strings.Bounded and the formal package parameter named Bounded.

5/3 {*AI95-00285-01*} {*AI95-00301-01*} {*AI05-0092-1*} The specification of package Wide_Text_IO.-Wide_Unbounded_IO is the same as that for Text_IO.Unbounded_IO, except that any occurrence of Unbounded_String is replaced by Unbounded_Wide_String, and any occurrence of package Unbounded is replaced by Wide_Unbounded. The specification of package Wide_Wide_Text_IO.Wide_Wide_-Unbounded_IO is the same as that for Text_IO.Unbounded_IO, except that any occurrence of Unbounded_String is replaced by Unbounded_Wide_Wide_String, and any occurrence of package Unbounded is replaced by Wide_Wide_Unbounded.

Extensions to Ada 83

5.a Support for Wide_Character and Wide_String I/O is new in Ada 95.

Extensions to Ada 95

5.b/2 {*AI95-00285-01*} Package Wide_Wide_Text_IO is new. Be glad it wasn't called Double_Wide_Text_IO (for use in trailer parks) or Really_Wide_Text_IO.

5.c/2 {*AI95-00301-01*} Packages Wide_Text_IO.Wide_Unbounded_IO and Wide_Wide_Text_IO.Wide_Wide_Unbounded_IO are also new.

5.d/2 {*AI95-00428-01*} Packages Wide_Text_IO.Wide_Bounded_IO and Wide_Wide_Text_IO.Wide_Wide_Bounded_IO are new as well.

Wording Changes from Ada 2005

5.e/3 {*AI05-0092-1*} **Correction**: Corrected the names of various entities in the above description. Since the previously named entities don't exist and the intent is obvious, this is just considered a presentation change.

A.12 Stream Input-Output

1/2 {*AI95-00285-01*} The packages Streams.Stream_IO, Text_IO.Text_Streams, Wide_Text_IO.Text_Streams, and Wide_Wide_Text_IO.Text_Streams provide stream-oriented operations on files.

Wording Changes from Ada 95

1.a/2 {*AI95-00285-01*} Included package Wide_Wide_Text_IO.Text_Streams in this description.

A.12.1 The Package Streams.Stream_IO

1 [The subprograms in the child package Streams.Stream_IO provide control over stream files. Access to a stream file is either sequential, via a call on Read or Write to transfer an array of stream elements, or positional (if supported by the implementation for the given file), by specifying a relative index for an

element. Since a stream file can be converted to a Stream_Access value, calling stream-oriented attribute subprograms of different element types with the same Stream_Access value provides heterogeneous input-output.] See 13.13 for a general discussion of streams.

Static Semantics

{*8652/0055*} {*AI95-00026-01*} The elements of a stream file are stream elements. If positioning is supported for the specified external file, a current index and current size are maintained for the file as described in A.8. If positioning is not supported, a current index is not maintained, and the current size is implementation defined. 1.1/1

Implementation defined: Current size for a stream file for which positioning is not supported. 1.a.1/1

The library package Streams.Stream_IO has the following declaration: 2

```
{AI05-0283-1} {AI12-0302-1} with Ada.IO_Exceptions;                                    3/5
package Ada.Streams.Stream_IO
    with Preelaborate, Global => in out synchronized is

    type Stream_Access is access all Root_Stream_Type'Class;                           4

{AI12-0102-1} {AI12-0399-1}     type File_Type is limited private                      5/5
        with Preelaborable_Initialization;

    type File_Mode is (In_File, Out_File, Append_File);                                6

    type    Count          is range 0 .. implementation-defined;                       7
    subtype Positive_Count is Count range 1 .. Count'Last;
      -- Index into file, in stream elements.

    procedure Create (File : in out File_Type;                                         8
                      Mode : in File_Mode := Out_File;
                      Name : in String    := "";
                      Form : in String    := "");

    procedure Open (File : in out File_Type;                                           9
                    Mode : in File_Mode;
                    Name : in String;
                    Form : in String := "");

    procedure Close  (File : in out File_Type);                                        10
    procedure Delete (File : in out File_Type);
    procedure Reset  (File : in out File_Type; Mode : in File_Mode);
    procedure Reset  (File : in out File_Type);

    function Mode (File : in File_Type) return File_Mode;                              11
    function Name (File : in File_Type) return String;
    function Form (File : in File_Type) return String;

    function Is_Open     (File : in File_Type) return Boolean;                         12
    function End_Of_File (File : in File_Type) return Boolean;

    function Stream (File : in File_Type) return Stream_Access;                        13
        -- Return stream access for use with T'Input and T'Output

This paragraph was deleted.                                                            14/1

{AI12-0302-1}     -- Read array of stream elements from file                           15/5
    procedure Read (File : in  File_Type;
                    Item : out Stream_Element_Array;
                    Last : out Stream_Element_Offset;
                    From : in  Positive_Count)
        with Global => overriding in out File;

{AI12-0302-1}     procedure Read (File : in  File_Type;                                16/5
                    Item : out Stream_Element_Array;
                    Last : out Stream_Element_Offset)
        with Global => overriding in out File;

This paragraph was deleted.                                                            17/1
```

```
18/5    {AI12-0302-1}      -- Write array of stream elements into file
           procedure Write (File : in File_Type;
                            Item : in Stream_Element_Array;
                            To   : in Positive_Count)
              with Global => overriding in out File;
19/5    {AI12-0302-1}      procedure Write (File : in File_Type;
                               Item : in Stream_Element_Array)
              with Global => overriding in out File;
20/1    This paragraph was deleted.
21         -- Operations on position within file
22/5    {AI12-0302-1}      procedure Set_Index(File : in File_Type; To : in Positive_Count)
              with Global => overriding in out File;
23         function Index(File : in File_Type) return Positive_Count;
           function Size (File : in File_Type) return Count;
24         procedure Set_Mode(File : in out File_Type; Mode : in File_Mode);
25/1    {8652/0051} {AI95-00057-01}      procedure Flush(File : in File_Type);
26         -- exceptions
           Status_Error : exception renames IO_Exceptions.Status_Error;
           Mode_Error   : exception renames IO_Exceptions.Mode_Error;
           Name_Error   : exception renames IO_Exceptions.Name_Error;
           Use_Error    : exception renames IO_Exceptions.Use_Error;
           Device_Error : exception renames IO_Exceptions.Device_Error;
           End_Error    : exception renames IO_Exceptions.End_Error;
           Data_Error   : exception renames IO_Exceptions.Data_Error;
26.1/5  {AI12-0021-1}     package Wide_File_Names is
26.2/5        -- File management
26.3/5        procedure Create (File : in out File_Type;
                                Mode : in File_Mode := Out_File;
                                Name : in Wide_String := "";
                                Form : in Wide_String := "");
26.4/5        procedure Open (File : in out File_Type;
                              Mode : in File_Mode;
                              Name : in Wide_String;
                              Form : in Wide_String := "");
26.5/5        function Name (File : in File_Type) return Wide_String;
26.6/5        function Form (File : in File_Type) return Wide_String;
26.7/5     end Wide_File_Names;
26.8/5  {AI12-0021-1}     package Wide_Wide_File_Names is
26.9/5        -- File management
26.10/5       procedure Create (File : in out File_Type;
                                Mode : in File_Mode := Out_File;
                                Name : in Wide_Wide_String := "";
                                Form : in Wide_Wide_String := "");
26.11/5       procedure Open (File : in out File_Type;
                              Mode : in File_Mode;
                              Name : in Wide_Wide_String;
                              Form : in Wide_Wide_String := "");
26.12/5       function Name (File : in File_Type) return Wide_Wide_String;
26.13/5       function Form (File : in File_Type) return Wide_Wide_String;
26.14/5    end Wide_Wide_File_Names;
27      private
           ... -- not specified by the language
        end Ada.Streams.Stream_IO;
```

27.1/2 {*AI95-00360-01*} The type File_Type needs finalization (see 7.6).

{*AI95-00283-01*} {*AI12-0130-1*} {*AI12-0449-1*} The subprograms given in A.8.2 for the control of external files (Create, Open, Close, Delete, Reset, Mode, Name, Form, Is_Open, and Flush) are available for stream files. 28/5

{*AI95-00283-01*} The End_Of_File function: 28.1/2

- Propagates Mode_Error if the mode of the file is not In_File; 28.2/2
- {*AI05-0264-1*} If positioning is supported for the given external file, the function returns True if the current index exceeds the size of the external file; otherwise, it returns False; 28.3/3
- {*AI05-0264-1*} If positioning is not supported for the given external file, the function returns True if no more elements can be read from the given file; otherwise, it returns False. 28.4/3

{*8652/0055*} {*AI95-00026-01*} {*AI95-00085-01*} The Set_Mode procedure sets the mode of the file. If the new mode is Append_File, the file is positioned to its end; otherwise, the position in the file is unchanged. 28.5/2

This paragraph was deleted.{*8652/0055*} {*AI95-00026-01*} {*AI12-0130-1*} 28.6/4

{*8652/0056*} {*AI95-00001-01*} The Stream function returns a Stream_Access result from a File_Type object, thus allowing the stream-oriented attributes Read, Write, Input, and Output to be used on the same file for multiple types. Stream propagates Status_Error if File is not open. 29/1

{*AI95-00256-01*} The procedures Read and Write are equivalent to the corresponding operations in the package Streams. Read propagates Mode_Error if the mode of File is not In_File. Write propagates Mode_Error if the mode of File is not Out_File or Append_File. The Read procedure with a Positive_Count parameter starts reading at the specified index. The Write procedure with a Positive_Count parameter starts writing at the specified index. For a file that supports positioning, Read without a Positive_Count parameter starts reading at the current index, and Write without a Positive_Count parameter starts writing at the current index. 30/2

{*8652/0055*} {*AI95-00026-01*} The Size function returns the current size of the file. 30.1/1

{*8652/0055*} {*AI95-00026-01*} The Index function returns the current index. 31/1

This paragraph was deleted. 31.a/1

The Set_Index procedure sets the current index to the specified value. 32

{*8652/0055*} {*AI95-00026-01*} If positioning is supported for the external file, the current index is maintained as follows: 32.1/1

- {*8652/0055*} {*AI95-00026-01*} For Open and Create, if the Mode parameter is Append_File, the current index is set to the current size of the file plus one; otherwise, the current index is set to one. 32.2/1
- {*8652/0055*} {*AI95-00026-01*} For Reset, if the Mode parameter is Append_File, or no Mode parameter is given and the current mode is Append_File, the current index is set to the current size of the file plus one; otherwise, the current index is set to one. 32.3/1
- {*8652/0055*} {*AI95-00026-01*} For Set_Mode, if the new mode is Append_File, the current index is set to current size plus one; otherwise, the current index is unchanged. 32.4/1
- {*8652/0055*} {*AI95-00026-01*} For Read and Write without a Positive_Count parameter, the current index is incremented by the number of stream elements read or written. 32.5/1
- {*8652/0055*} {*AI95-00026-01*} For Read and Write with a Positive_Count parameter, the value of the current index is set to the value of the Positive_Count parameter plus the number of stream elements read or written. 32.6/1

33 If positioning is not supported for the given file, then a call of Index or Set_Index propagates Use_Error. Similarly, a call of Read or Write with a Positive_Count parameter propagates Use_Error.

33.a/2 **Implementation Note:** {*AI95-00085-01*} It is permissible for an implementation to implement mode Append_File using the Unix append mode (the O_APPEND bit). Such an implementation does not support positioning when the mode is Append_File, and therefore the operations listed above must raise Use_Error. This is acceptable as there is no requirement that any particular file support positioning; therefore it is acceptable that a file support positioning when opened with mode Out_File, and the same file not support positioning when opened with mode Append_File. But it is not acceptable for a file to support positioning (by allowing the above operations), but to do something other than the defined semantics (that is, always write at the end, even when explicitly commanded to write somewhere else).

Paragraphs 34 through 36 were deleted.

Erroneous Execution

36.1/1 {*8652/0056*} {*AI95-00001-01*} If the File_Type object passed to the Stream function is later closed or finalized, and the stream-oriented attributes are subsequently called (explicitly or implicitly) on the Stream_Access value returned by Stream, execution is erroneous. This rule applies even if the File_Type object was opened again after it had been closed.

36.a.1/1 **Reason:** These rules are analogous to the rule for the result of the Current_Input, Current_Output, and Current_Error functions. These rules make it possible to represent a value of (some descendant of) Root_Stream_Type which represents a file as an access value, with a null value corresponding to a closed file.

Inconsistencies With Ada 95

36.a/3 {*AI95-00283-01*} {*AI05-0005-1*} **Amendment Correction:** The description of the subprograms for managing files was corrected so that they do not require truncation of the external file — a stream file is not a sequential file. An Ada 95 program that expects truncation of the stream file might not work under Ada 2005. Note that the Ada 95 standard was ambiguous on this point (the normative wording seemed to require truncation, but didn't explain where; the AARM notes seemed to expect behavior like Direct_IO), and implementations varied widely. Therefore, as a practical matter, code that depends on stream truncation might not work even in Ada 95; deleting the file before opening it provides truncation that works in both Ada 95 and Ada 2005.

Incompatibilities With Ada 95

36.b/2 {*AI95-00360-01*} **Amendment Correction:** Stream_IO.File_Type is defined to need finalization. If the restriction No_Nested_Finalization (see D.7) applies to the partition, and File_Type does not have a controlled part, it will not be allowed in local objects in Ada 2005 whereas it would be allowed in original Ada 95. Such code is not portable, as another Ada compiler may have a controlled part in File_Type, and thus would be illegal.

Wording Changes from Ada 95

36.c/2 {*8652/0051*} {*AI95-00057-01*} **Corrigendum:** Corrected the parameter mode of Flush; otherwise it could not be used on Standard_Output.

36.d/2 {*8652/0055*} {*AI95-00026-01*} {*AI95-00256-01*} **Corrigendum:** Added wording to describe the effects of the various operations on the current index. The Amendment adds an explanation of the use of current index for Read and Write.

36.e/2 {*8652/0056*} {*AI95-00001-01*} **Corrigendum:** Clarified that Stream can raise Status_Error, and clarified that using a Stream_Access whose file has been closed is erroneous.

36.f/2 {*AI95-00085-01*} Clarified that Set_Mode can be called with the current mode.

Extensions to Ada 2005

36.g/3 {*AI05-0283-1*} Package Ada.Streams.Stream_IO is now preelaborated, allowing it to be used in more contexts (including in distributed systems). Note that it is *not* a remote types package; File_Type objects cannot be passed between partitions.

Incompatibilities With Ada 2012

36.h/5 {*AI12-0021-1*} The Wide_File_Names and Wide_Wide_File_Names nested packages are newly added to Ada.Stream_IO. Therefore, a use clause conflict is possible; see the introduction of Annex A for more on this topic.

Extensions to Ada 2012

{*AI12-0102-1*} **Corrigendum:** Type Ada.Streams.Stream_IO.File_Type now has preelaborable initialization. This allows declaring library-level file objects in preelaborable packages (an oversight from the change above). 36.i/4

Wording Changes from Ada 2012

{*AI12-0130-1*} **Corrigendum:** The definition of the Flush procedure was moved to A.8.2, so that it could be shared by all of the I/O packages. 36.j/4

A.12.2 The Package Text_IO.Text_Streams

The package Text_IO.Text_Streams provides a function for treating a text file as a stream. 1

Static Semantics

The library package Text_IO.Text_Streams has the following declaration: 2

```
{AI12-0302-1} with Ada.Streams;                                                    3/5
package Ada.Text_IO.Text_Streams
   with Global => in out synchronized is
   type Stream_Access is access all Streams.Root_Stream_Type'Class;

   function Stream (File : in File_Type) return Stream_Access;                     4
end Ada.Text_IO.Text_Streams;
```

The Stream function has the same effect as the corresponding function in Streams.Stream_IO. 5

NOTE 1 The ability to obtain a stream for a text file allows Current_Input, Current_Output, and Current_Error to be processed with the functionality of streams, including the mixing of text and binary input-output, and the mixing of binary input-output for different types. 6

NOTE 2 Performing operations on the stream associated with a text file does not affect the column, line, or page counts. 7

A.12.3 The Package Wide_Text_IO.Text_Streams

The package Wide_Text_IO.Text_Streams provides a function for treating a wide text file as a stream. 1

Static Semantics

The library package Wide_Text_IO.Text_Streams has the following declaration: 2

```
{AI12-0302-1} with Ada.Streams;                                                    3/5
package Ada.Wide_Text_IO.Text_Streams
   with Global => in out synchronized is
   type Stream_Access is access all Streams.Root_Stream_Type'Class;

   function Stream (File : in File_Type) return Stream_Access;                     4
end Ada.Wide_Text_IO.Text_Streams;
```

The Stream function has the same effect as the corresponding function in Streams.Stream_IO. 5

A.12.4 The Package Wide_Wide_Text_IO.Text_Streams

{*AI95-00285-01*} The package Wide_Wide_Text_IO.Text_Streams provides a function for treating a wide wide text file as a stream. 1/2

Static Semantics

{*AI95-00285-01*} The library package Wide_Wide_Text_IO.Text_Streams has the following declaration: 2/2

```
{AI12-0302-1} with Ada.Streams;                                                    3/5
package Ada.Wide_Wide_Text_IO.Text_Streams
   with Global => in out synchronized is
   type Stream_Access is access all Streams.Root_Stream_Type'Class;
```

4/2
```
      function Stream (File : in File_Type) return Stream_Access;
   end Ada.Wide_Wide_Text_IO.Text_Streams;
```

5/2 {*AI95-00285-01*} The Stream function has the same effect as the corresponding function in Streams.Stream_IO.

Extensions to Ada 95

5.a/2 {*AI95-00285-01*} Package Wide_Wide_Text_IO.Text_Streams is new.

A.13 Exceptions in Input-Output

1/5 {*AI12-0445-1*} The package IO_Exceptions defines the exceptions used by the predefined input-output packages.

Static Semantics

2 The library package IO_Exceptions has the following declaration:

3/5
```
   {AI12-0414-1} package Ada.IO_Exceptions
      with Pure is
```

4
```
      Status_Error : exception;
      Mode_Error   : exception;
      Name_Error   : exception;
      Use_Error    : exception;
      Device_Error : exception;
      End_Error    : exception;
      Data_Error   : exception;
      Layout_Error : exception;
```

5
```
   end Ada.IO_Exceptions;
```

6 If more than one error condition exists, the corresponding exception that appears earliest in the following list is the one that is propagated.

7 The exception Status_Error is propagated by an attempt to operate upon a file that is not open, and by an attempt to open a file that is already open.

8 The exception Mode_Error is propagated by an attempt to read from, or test for the end of, a file whose current mode is Out_File or Append_File, and also by an attempt to write to a file whose current mode is In_File. In the case of Text_IO, the exception Mode_Error is also propagated by specifying a file whose current mode is Out_File or Append_File in a call of Set_Input, Skip_Line, End_Of_Line, Skip_Page, or End_Of_Page; and by specifying a file whose current mode is In_File in a call of Set_Output, Set_Line_Length, Set_Page_Length, Line_Length, Page_Length, New_Line, or New_Page.

9 The exception Name_Error is propagated by a call of Create or Open if the string given for the parameter Name does not allow the identification of an external file. For example, this exception is propagated if the string is improper, or, alternatively, if either none or more than one external file corresponds to the string.

10 The exception Use_Error is propagated if an operation is attempted that is not possible for reasons that depend on characteristics of the external file. For example, this exception is propagated by the procedure Create, among other circumstances, if the given mode is Out_File but the form specifies an input only device, if the parameter Form specifies invalid access rights, or if an external file with the given name already exists and overwriting is not allowed.

11 The exception Device_Error is propagated if an input-output operation cannot be completed because of a malfunction of the underlying system.

12 The exception End_Error is propagated by an attempt to skip (read past) the end of a file.

The exception Data_Error can be propagated by the procedure Read (or by the Read attribute) if the element read cannot be interpreted as a value of the required subtype. This exception is also propagated by a procedure Get (defined in the package Text_IO) if the input character sequence fails to satisfy the required syntax, or if the value input does not belong to the range of the required subtype. 13

The exception Layout_Error is propagated (in text input-output) by Col, Line, or Page if the value returned exceeds Count'Last. The exception Layout_Error is also propagated on output by an attempt to set column or line numbers in excess of specified maximum line or page lengths, respectively (excluding the unbounded cases). It is also propagated by an attempt to Put too many characters to a string. 14

{*AI05-0262-1*} These exceptions are also propagated by various other language-defined packages and operations, see the definition of those entities for other reasons that these exceptions are propagated. 14.1/3

Reason: {*AI05-0299-1*} This subclause is based in Ada 95. Later versions of Ada (starting with Technical Corrigendum 1) have added a number of additional places and reasons that cause these exceptions. In particular, TC1 says that stream attributes need to raise End_Error in some circumstances; Amendment 1 adds Ada.Directories and a number of new places and reasons that Name_Error and Use_Error are raised. There are more. We don't want to try to update this text (or even this note!) for every possible reason and place that might raise one of these exceptions, so we add this blanket statement. 14.a/3

Documentation Requirements

The implementation shall document the conditions under which Name_Error, Use_Error and Device_Error are propagated. 15

Documentation Requirement: The conditions under which Io_Exceptions.Name_Error, Io_Exceptions.Use_Error, and Io_Exceptions.Device_Error are propagated. 15.a/2

Implementation Permissions

{*AI12-0444-1*} When the associated check is complex, it is optional to propagate Data_Error as part of a procedure Read (or the Read attribute) when the value read cannot be interpreted as a value of the required subtype. 16/5

Ramification: An example where the implementation may choose not to perform the check is an enumeration type with a representation clause with “holes” in the range of internal codes. 16.a

Erroneous Execution

[If the element read by the procedure Read (or by the Read attribute) cannot be interpreted as a value of the required subtype, but this is not detected and Data_Error is not propagated, then the resulting value can be abnormal, and subsequent references to the value can lead to erroneous execution, as explained in 13.9.1.] 17

A.14 File Sharing

Dynamic Semantics

It is not specified by the language whether the same external file can be associated with more than one file object. If such sharing is supported by the implementation, the following effects are defined: 1

- Operations on one text file object do not affect the column, line, and page numbers of any other file object. 2
- *This paragraph was deleted.*{*8652/0057*} {*AI95-00050-01*} 3/1
- For direct and stream files, the current index is a property of each file object; an operation on one file object does not affect the current index of any other file object. 4
- For direct and stream files, the current size of the file is a property of the external file. 5

6 All other effects are identical.

Wording Changes from Ada 95

6.a/2 {*8652/0057*} {*AI95-00050-01*} **Corrigendum:** Removed the incorrect statement that the external files associated with the standard input, standard output, and standard error files are distinct.

A.15 The Package Command_Line

1 The package Command_Line allows a program to obtain the values of its arguments and to set the exit status code to be returned on normal termination.

1.a/2 **Implementation defined:** The meaning of Argument_Count, Argument, and Command_Name for package Command_Line. The bounds of type Command_Line.Exit_Status.

Static Semantics

2 The library package Ada.Command_Line has the following declaration:

3/5
```
{AI12-0241-1} package Ada.Command_Line
  with Preelaborate, Nonblocking, Global => in out synchronized is
```

4
```
   function Argument_Count return Natural;
```

5
```
   function Argument (Number : in Positive) return String;
```

6
```
   function Command_Name return String;
```

7
```
   type Exit_Status is implementation-defined integer type;
```

8
```
   Success : constant Exit_Status;
   Failure : constant Exit_Status;
```

9
```
   procedure Set_Exit_Status (Code : in Exit_Status);
```

10
```
private
   ... -- not specified by the language
end Ada.Command_Line;
```

11
```
function Argument_Count return Natural;
```

12/3 {*AI05-0264-1*} If the external execution environment supports passing arguments to a program, then Argument_Count returns the number of arguments passed to the program invoking the function. Otherwise, it returns 0. The meaning of "number of arguments" is implementation defined.

13
```
function Argument (Number : in Positive) return String;
```

14/5 {*AI12-0259-1*} If the external execution environment supports passing arguments to a program, then Argument returns an implementation-defined value with lower bound 1 corresponding to the argument at relative position Number. If Number is outside the range 1..Argument_Count, then Constraint_Error is propagated.

14.a **Ramification:** If the external execution environment does not support passing arguments to a program, then Argument(N) for any N will raise Constraint_Error, since Argument_Count is 0.

15
```
function Command_Name return String;
```

16/5 {*AI05-0264-1*} {*AI12-0259-1*} If the external execution environment supports passing arguments to a program, then Command_Name returns an implementation-defined value with lower bound 1 corresponding to the name of the command invoking the program; otherwise, Command_Name returns the null string.

```
type Exit_Status is implementation-defined integer type;
```
16.1/1

The type Exit_Status represents the range of exit status values supported by the external execution environment. The constants Success and Failure correspond to success and failure, respectively. 17

```
procedure Set_Exit_Status (Code : in Exit_Status);
```
18

If the external execution environment supports returning an exit status from a program, then Set_Exit_Status sets Code as the status. Normal termination of a program returns as the exit status the value most recently set by Set_Exit_Status, or, if no such value has been set, then the value Success. If a program terminates abnormally, the status set by Set_Exit_Status is ignored, and an implementation-defined exit status value is set. 19

If the external execution environment does not support returning an exit value from a program, then Set_Exit_Status does nothing. 20

Implementation Permissions

An alternative declaration is allowed for package Command_Line if different functionality is appropriate for the external execution environment. 21

NOTE Argument_Count, Argument, and Command_Name correspond to the C language's argc, argv[n] (for n>0) and argv[0], respectively. 22

To be honest: The correspondence of Argument_Count to argc is not direct — argc would be one more than Argument_Count, since the argc count includes the command name, whereas Argument_Count does not. 22.a

Extensions to Ada 83

{*AI05-0299-1*} This subclause is new in Ada 95. 22.b/3

Wording Changes from Ada 2012

{*AI12-0259-1*} **Correction:** Defined the lower bound of functions Argument and Command_Name. This could be inconsistent if someone depended on the lower bound of these routines (and it wasn't 1), but such code was never portable (even to later versions of the same implementation). Thus we don't document it as an inconsistency. 22.c/5

A.15.1 The Packages Wide_Command_Line and Wide_Wide_Command_Line

{*AI12-0021-1*} The packages Wide_Command_Line and Wide_Wide_Command_Line allow a program to obtain the values of its arguments and to set the exit status code to be returned on normal termination. 1/5

Static Semantics

{*AI12-0021-1*} The specification of package Wide_Command_Line is the same as for Command_Line, except that each occurrence of String is replaced by Wide_String. 2/5

{*AI12-0021-1*} The specification of package Wide_Wide_Command_Line is the same as for Command_Line, except that each occurrence of String is replaced by Wide_Wide_String. 3/5

Extensions to Ada 2012

{*AI12-0021-1*} These packages are new. 3.a/5

A.16 The Package Directories

{*AI95-00248-01*} The package Directories provides operations for manipulating files and directories, and their names. 1/2

1.a/3 **Discussion:** {*AI05-0299-1*} The notes for this subclause contain the expected interpretations of some of the operations on various target systems. “Unix” refers to the UNIX® operating system, and in most cases also covers Unix-like systems such as Linux and POSIX. “Windows®” refers to the Microsoft® Windows® 2000 operating system and usually also covers most other versions that use the Win32 API.

Static Semantics

2/2 {*AI95-00248-01*} The library package Directories has the following declaration:

3/5
```
{AI12-0302-1} with Ada.IO_Exceptions;
with Ada.Calendar;
package Ada.Directories
   with Global => in out synchronized is
```

4/2
```
   -- Directory and file operations:
```

5/2
```
   function Current_Directory return String;
```

6/2
```
   procedure Set_Directory (Directory : in String);
```

7/2
```
   procedure Create_Directory (New_Directory : in String;
                               Form          : in String := "");
```

8/2
```
   procedure Delete_Directory (Directory : in String);
```

9/2
```
   procedure Create_Path (New_Directory : in String;
                          Form          : in String := "");
```

10/2
```
   procedure Delete_Tree (Directory : in String);
```

11/2
```
   procedure Delete_File (Name : in String);
```

12/2
```
   procedure Rename (Old_Name, New_Name : in String);
```

13/2
```
   procedure Copy_File (Source_Name,
                        Target_Name : in String;
                        Form        : in String := "");
```

14/2
```
   -- File and directory name operations:
```

15/5
```
   {AI12-0241-1}    function Full_Name (Name : in String) return String
         with Nonblocking;
```

16/5
```
   {AI12-0241-1}    function Simple_Name (Name : in String) return String
         with Nonblocking;
```

17/5
```
   {AI12-0241-1}    function Containing_Directory (Name : in String) return String
         with Nonblocking;
```

18/5
```
   {AI12-0241-1}    function Extension (Name : in String) return String
         with Nonblocking;
```

19/5
```
   {AI12-0241-1}    function Base_Name (Name : in String) return String
         with Nonblocking;
```

20/5
```
   {AI12-0241-1}    function Compose (Containing_Directory : in String := "";
                        Name                : in String;
                        Extension           : in String := "") return String
         with Nonblocking;
```

20.1/3
```
   {AI05-0049-1}    type Name_Case_Kind is
         (Unknown, Case_Sensitive, Case_Insensitive, Case_Preserving);
```

20.2/3
```
   {AI05-0049-1}    function Name_Case_Equivalence (Name : in String) return
   Name_Case_Kind;
```

21/2
```
   -- File and directory queries:
```

22/2
```
   type File_Kind is (Directory, Ordinary_File, Special_File);
```

23/2
```
   type File_Size is range 0 .. implementation-defined;
```

24/2
```
   function Exists (Name : in String) return Boolean;
```

25/2
```
   function Kind (Name : in String) return File_Kind;
```

26/2
```
   function Size (Name : in String) return File_Size;
```

27/2
```
   function Modification_Time (Name : in String) return Ada.Calendar.Time;
```

28/2
```
   -- Directory searching:
```

```
   type Directory_Entry_Type is limited private;                                    29/2
   type Filter_Type is array (File_Kind) of Boolean;                                30/2
   type Search_Type is limited private;                                             31/2
   procedure Start_Search (Search    : in out Search_Type;                          32/2
                           Directory : in String;
                           Pattern   : in String;
                           Filter    : in Filter_Type := (others => True));
   procedure End_Search (Search : in out Search_Type);                               33/2
   function More_Entries (Search : in Search_Type) return Boolean;                  34/2
   procedure Get_Next_Entry (Search : in out Search_Type;                           35/2
                             Directory_Entry : out Directory_Entry_Type);
{AI12-0286-1}    procedure Search (                                                  36/5
      Directory : in String;
      Pattern   : in String;
      Filter    : in Filter_Type := (others => True);
      Process   : not null access procedure (
          Directory_Entry : in Directory_Entry_Type))
      with Allows_Exit;
   -- Operations on Directory Entries:                                              37/2
   function Simple_Name (Directory_Entry : in Directory_Entry_Type)                 38/2
       return String;
   function Full_Name (Directory_Entry : in Directory_Entry_Type)                   39/2
       return String;
   function Kind (Directory_Entry : in Directory_Entry_Type)                        40/2
       return File_Kind;
   function Size (Directory_Entry : in Directory_Entry_Type)                        41/2
       return File_Size;
   function Modification_Time (Directory_Entry : in Directory_Entry_Type)           42/2
       return Ada.Calendar.Time;
   Status_Error : exception renames Ada.IO_Exceptions.Status_Error;                 43/2
   Name_Error   : exception renames Ada.IO_Exceptions.Name_Error;
   Use_Error    : exception renames Ada.IO_Exceptions.Use_Error;
   Device_Error : exception renames Ada.IO_Exceptions.Device_Error;
{AI05-0092-1} private                                                               44/3
   ... -- not specified by the language
end Ada.Directories;
```

{*AI95-00248-01*} External files may be classified as directories, special files, or ordinary files. A *directory* is an external file that is a container for files on the target system. A *special file* is an external file that cannot be created or read by a predefined Ada input-output package. External files that are not special files or directories are called *ordinary files*. 45/2

Ramification: A directory is an external file, although it may not have a name on some targets. A directory is not a special file, as it can be created and read by Directories. 45.a/2

Discussion: Devices and soft links are examples of special files on Windows® and Unix. 45.b/2

Even if an implementation provides a package to create and read soft links, such links are still special files. 45.c/2

{*AI95-00248-01*} A *file name* is a string identifying an external file. Similarly, a *directory name* is a string identifying a directory. The interpretation of file names and directory names is implementation defined. 46/2

Implementation defined: The interpretation of file names and directory names. 46.a/2

{*AI95-00248-01*} The *full name* of an external file is a full specification of the name of the file. If the external environment allows alternative specifications of the name (for example, abbreviations), the full name should not use such alternatives. A full name typically will include the names of all of the directories that contain the item. The *simple name* of an external file is the name of the item, not including 47/2

any containing directory names. Unless otherwise specified, a file name or directory name parameter in a call to a predefined Ada input-output subprogram can be a full name, a simple name, or any other form of name supported by the implementation.

47.a/2 **Discussion:** The full name on Unix is a complete path to the root. For Windows®, the full name includes a complete path, as well as a disk name ("C:") or network share name. For both systems, the simple name is the part of the name following the last '/' (or '\' for Windows®). For example, in the name "/usr/randy/ada-directories.ads", "ada-directories.ads" is the simple name.

47.b/2 **Ramification:** It is possible for a file or directory name to be neither a full name nor a simple name. For instance, the Unix name "../parent/myfile" is neither a full name nor a simple name.

47.1/5 {*AI12-0337-1*} A *root directory* is a directory that has no containing directory.

47.c/5 **Implementation Note:** For Unix and Unix-like systems, "/" is the root. For Windows®, "C:\" and "\\Computer\Share" are roots.

48/2 {*AI95-00248-01*} The *default directory* is the directory that is used if a directory or file name is not a full name (that is, when the name does not fully identify all of the containing directories).

48.a/2 **Discussion:** The default directory is the one maintained by the familiar “cd” command on Unix and Windows®. Note that Windows® maintains separate default directories for each disk drive; implementations should use the natural implementation.

49/2 {*AI95-00248-01*} A *directory entry* is a single item in a directory, identifying a single external file (including directories and special files).

50/2 {*AI95-00248-01*} For each function that returns a string, the lower bound of the returned value is 1.

51/2 {*AI95-00248-01*} The following file and directory operations are provided:

52/2
```
function Current_Directory return String;
```

53/2 Returns the full directory name for the current default directory. The name returned shall be suitable for a future call to Set_Directory. The exception Use_Error is propagated if a default directory is not supported by the external environment.

54/2
```
procedure Set_Directory (Directory : in String);
```

55/2 Sets the current default directory. The exception Name_Error is propagated if the string given as Directory does not identify an existing directory. The exception Use_Error is propagated if the external environment does not support making Directory (in the absence of Name_Error) a default directory.

56/2
```
procedure Create_Directory (New_Directory : in String;
                           Form          : in String := "");
```

57/2 Creates a directory with name New_Directory. The Form parameter can be used to give system-dependent characteristics of the directory; the interpretation of the Form parameter is implementation defined. A null string for Form specifies the use of the default options of the implementation of the new directory. The exception Name_Error is propagated if the string given as New_Directory does not allow the identification of a directory. The exception Use_Error is propagated if the external environment does not support the creation of a directory with the given name (in the absence of Name_Error) and form.

58/2
```
procedure Delete_Directory (Directory : in String);
```

59/3 {*AI05-0231-1*} Deletes an existing empty directory with name Directory. The exception Name_Error is propagated if the string given as Directory does not identify an existing directory. The exception Use_Error is propagated if the directory is not empty or the external environment does not support the deletion of the directory with the given name (in the absence of Name_Error).

```
procedure Create_Path (New_Directory : in String;                    60/2
                       Form          : in String := "");
```

{*AI05-0271-1*} Creates zero or more directories with name New_Directory. Each nonexistent directory named by New_Directory is created.[For example, on a typical Unix system, Create_Path ("/usr/me/my"); would create directory "me" in directory "usr", then create directory "my" in directory "me".] The Form parameter can be used to give system-dependent characteristics of the directory; the interpretation of the Form parameter is implementation defined. A null string for Form specifies the use of the default options of the implementation of the new directory. The exception Name_Error is propagated if the string given as New_Directory does not allow the identification of any directory. The exception Use_Error is propagated if the external environment does not support the creation of any directories with the given name (in the absence of Name_Error) and form. If Use_Error is propagated, it is unspecified whether a portion of the directory path is created. 61/3

```
procedure Delete_Tree (Directory : in String);                       62/2
```

Deletes an existing directory with name Directory. The directory and all of its contents (possibly including other directories) are deleted. The exception Name_Error is propagated if the string given as Directory does not identify an existing directory. The exception Use_Error is propagated if the external environment does not support the deletion of the directory or some portion of its contents with the given name (in the absence of Name_Error). If Use_Error is propagated, it is unspecified whether a portion of the contents of the directory is deleted. 63/2

```
procedure Delete_File (Name : in String);                            64/2
```

Deletes an existing ordinary or special file with name Name. The exception Name_Error is propagated if the string given as Name does not identify an existing ordinary or special external file. The exception Use_Error is propagated if the external environment does not support the deletion of the file with the given name (in the absence of Name_Error). 65/2

```
procedure Rename (Old_Name, New_Name : in String);                   66/2
```

{*AI05-0231-1*} Renames an existing external file (including directories) with name Old_Name to New_Name. The exception Name_Error is propagated if the string given as Old_Name does not identify an existing external file or if the string given as New_Name does not allow the identification of an external file. The exception Use_Error is propagated if the external environment does not support the renaming of the file with the given name (in the absence of Name_Error). In particular, Use_Error is propagated if a file or directory already exists with name New_Name. 67/3

Implementation Note: This operation is expected to work within a single directory, and implementers are encouraged to support it across directories on a single device. Copying files from one device to another is discouraged (that's what Copy_File is for). However, there is no requirement to detect file copying by the target system. If the target system has an API that gives that for “free”, it can be used. For Windows®, for instance, MoveFile can be used to implement Rename. 67.a/2

```
{AI05-0092-1} procedure Copy_File (Source_Name,                      68/3
                      Target_Name : in String;
                      Form        : in String := "");
```

{*AI05-0271-1*} Copies the contents of the existing external file with name Source_Name to an external file with name Target_Name. The resulting external file is a duplicate of the source external file. The Form parameter can be used to give system-dependent characteristics of the resulting external file; the interpretation of the Form parameter is implementation defined. Exception Name_Error is propagated if the string given as Source_Name does not identify an existing external ordinary or special file, or if the string given as Target_Name does not allow the identification of an external file. The exception Use_Error is propagated if the external 69/3

environment does not support creating the file with the name given by Target_Name and form given by Form, or copying of the file with the name given by Source_Name (in the absence of Name_Error). If Use_Error is propagated, it is unspecified whether a portion of the file is copied.

69.a/2 **Ramification:** Name_Error is always raised if Source_Name identifies a directory. It is up to the implementation whether special files can be copied, or if Use_Error will be raised.

70/2 {*AI95-00248-01*} The following file and directory name operations are provided:

71/2
```
function Full_Name (Name : in String) return String;
```

72/2 Returns the full name corresponding to the file name specified by Name. The exception Name_Error is propagated if the string given as Name does not allow the identification of an external file (including directories and special files).

72.a/2 **Discussion:** Full name means that no abbreviations are used in the returned name, and that it is a full specification of the name. Thus, for Unix and Windows®, the result should be a full path that does not contain any "." or ".." directories. Typically, the default directory is used to fill in any missing information.

73/2
```
function Simple_Name (Name : in String) return String;
```

74/5 {*AI12-0337-1*} {*AI12-0433-1*} Returns the simple name portion of the file name specified by Name. The simple name of a root directory is a name of the root itself. The exception Name_Error is propagated if the string given as Name does not allow the identification of an external file (including directories and special files).

74.a/5 **Discussion:** {*AI12-0337-1*} The result of Simple_Name corresponds to the result of the “basename” command on Linux and Unix. If the filename ends with a '/', and is not a root, then “basename” returns the part in front of all of the trailing '/'s. It returns a root intact. The null string is never returned. Similar rules should be used for Windows filenames.

75/2
```
function Containing_Directory (Name : in String) return String;
```

76/2 Returns the name of the containing directory of the external file (including directories) identified by Name. (If more than one directory can contain Name, the directory name returned is implementation defined.) The exception Name_Error is propagated if the string given as Name does not allow the identification of an external file. The exception Use_Error is propagated if the external file does not have a containing directory.

76.a/2 **Discussion:** This is purely a string manipulation function. If Name is not given as a full name, the containing directory probably won't be one, either. For example, if Containing_Directory ("..\AARM\RM-A-8") is called on Windows®, the result should be "..\AARM". If there is no path at all on the name, the result should be "." (which represents the current directory). Use Full_Name on the result of Containing_Directory if the full name is needed.

76.b/5 **Ramification:** {*AI12-0337-1*} Containing_Directory raises Use_Error when passed a string representing a root directory.

77/2
```
function Extension (Name : in String) return String;
```

78/2 Returns the extension name corresponding to Name. The extension name is a portion of a simple name (not including any separator characters), typically used to identify the file class. If the external environment does not have extension names, then the null string is returned. The exception Name_Error is propagated if the string given as Name does not allow the identification of an external file.

78.a/2 **Discussion:** For Unix and Windows®, the extension is the portion of the simple name following the rightmost period. For example, in the simple name "RM-A-8.html", the extension is "html".

79/2
```
function Base_Name (Name : in String) return String;
```

80/2 Returns the base name corresponding to Name. The base name is the remainder of a simple name after removing any extension and extension separators. The exception Name_Error is propagated if the string given as Name does not allow the identification of an external file (including directories and special files).

Discussion: For Unix and Windows®, the base name is the portion of the simple name preceding the rightmost period (except for the special directory names "." and "..", whose Base_Name is "." and ".."). For example, in the simple name "RM-A-8.html", the base name is "RM-A-8". 80.a/2

```
function Compose (Containing_Directory : in String := "";
                  Name                 : in String;
                  Extension            : in String := "") return String;
```
81/2

{*AI05-0264-1*} {*AI12-0337-1*} Returns the name of the external file with the specified Containing_Directory, Name, and Extension. If Extension is the null string, then Name is interpreted as a simple name; otherwise, Name is interpreted as a base name. The exception Name_Error is propagated if: 82/5

- the string given as Containing_Directory is not null and does not allow the identification of a directory; 82.1/5
- the string given as Extension is not null and is not a possible extension; 82.2/5
- the string given as Name is not a possible simple name (if Extension is null) or base name (if Extension is nonnull); or 82.3/5
- {*AI12-0337-1*} the string given as Name is a root directory, and Containing_Directory or Extension is nonnull. 82.4/5

Ramification: The above definition implies that if the Extension is null, for Unix and Windows® no '.' is added to Name. 82.a/2

Discussion: If Name is null, Name_Error should be raised, as nothing is not a possible simple name or base name. 82.b/2

Generally, Compose(Containing_Directory(F), Base_Name(F),Extension(F)) = F. However, this is not true on Unix or Windows® for file names that end with a '.'; Compose(Base_Name("Fooey."),Extension("Fooey.")) = "Fooey". This is not a problem for Windows®, as the names have the same meaning with or without the '.', but these are different names for Unix. Thus, care needs to be taken on Unix; if Extension is null, Base_Name should be avoided. (That's not usually a problem with file names generated by a program.) 82.c/2

```
{AI05-0049-1} function Name_Case_Equivalence (Name : in String) return
Name_Case_Kind;
```
82.5/3

{*AI05-0049-1*} {*AI05-0248-1*} Returns the file name equivalence rule for the directory containing Name. Raises Name_Error if Name is not a full name. Returns Case_Sensitive if file names that differ only in the case of letters are considered different names. If file names that differ only in the case of letters are considered the same name, then Case_Preserving is returned if names have the case of the file name used when a file is created; and Case_Insensitive is returned otherwise. Returns Unknown if the file name equivalence is not known. 82.6/3

Implementation Note: Unix, Linux, and their relatives are Case_Sensitive systems. Microsoft® Windows® is a Case_Preserving system (unless the rarely used POSIX mode is used). Ancient systems like CP/M and early MS-DOS were Case_Insensitive systems (file names were always in UPPER CASE). Unknown is provided in case it is impossible to tell (such as could be the case for network files). 82.c.1/3

{*AI95-00248-01*} The following file and directory queries and types are provided: 83/2

```
type File_Kind is (Directory, Ordinary_File, Special_File);
```
84/2

The type File_Kind represents the kind of file represented by an external file or directory. 85/2

```
type File_Size is range 0 .. implementation-defined;
```
86/2

The type File_Size represents the size of an external file. 87/2

Implementation defined: The maximum value for a file size in Directories. 87.a/2

```
function Exists (Name : in String) return Boolean;
```
88/2

Returns True if an external file represented by Name exists, and False otherwise. The exception Name_Error is propagated if the string given as Name does not allow the identification of an external file (including directories and special files). 89/2

90/2 `function Kind (Name : in String) return File_Kind;`

91/2 Returns the kind of external file represented by Name. The exception Name_Error is propagated if the string given as Name does not allow the identification of an existing external file.

92/2 `function Size (Name : in String) return File_Size;`

93/2 Returns the size of the external file represented by Name. The size of an external file is the number of stream elements contained in the file. If the external file is not an ordinary file, the result is implementation defined. The exception Name_Error is propagated if the string given as Name does not allow the identification of an existing external file. The exception Constraint_Error is propagated if the file size is not a value of type File_Size.

93.a/2 **Implementation defined:** The result for Directories.Size for a directory or special file.

93.b/2 **Discussion:** We allow raising Constraint_Error, so that an implementation for a system with 64-bit file sizes does not need to support full numerics on 64-bit integers just to implement this package. Of course, if 64-bit integers are available on such a system, they should be used when defining type File_Size.

94/2 `function Modification_Time (Name : in String) return Ada.Calendar.Time;`

95/2 Returns the time that the external file represented by Name was most recently modified. If the external file is not an ordinary file, the result is implementation defined. The exception Name_Error is propagated if the string given as Name does not allow the identification of an existing external file. The exception Use_Error is propagated if the external environment does not support reading the modification time of the file with the name given by Name (in the absence of Name_Error).

95.a/2 **Implementation defined:** The result for Directories.Modification_Time for a directory or special file.

96/2 {*AI95-00248-01*} The following directory searching operations and types are provided:

97/2 `type Directory_Entry_Type is limited private;`

98/2 The type Directory_Entry_Type represents a single item in a directory. These items can only be created by the Get_Next_Entry procedure in this package. Information about the item can be obtained from the functions declared in this package. A default-initialized object of this type is invalid; objects returned from Get_Next_Entry are valid.

99/2 `type Filter_Type is array (File_Kind) of Boolean;`

100/2 The type Filter_Type specifies which directory entries are provided from a search operation. If the Directory component is True, directory entries representing directories are provided. If the Ordinary_File component is True, directory entries representing ordinary files are provided. If the Special_File component is True, directory entries representing special files are provided.

101/2 `type Search_Type is limited private;`

102/2 The type Search_Type contains the state of a directory search. A default-initialized Search_Type object has no entries available (function More_Entries returns False). Type Search_Type needs finalization (see 7.6).

103/2
```
procedure Start_Search (Search    : in out Search_Type;
                        Directory : in String;
                        Pattern   : in String;
                        Filter    : in Filter_Type := (others => True));
```

104/3 {*AI05-0092-1*} {*AI05-0262-1*} Starts a search in the directory named by Directory for entries matching Pattern and Filter. Pattern represents a pattern for matching file names. If Pattern is the null string, all items in the directory are matched; otherwise, the interpretation of Pattern is implementation defined. Only items that match Filter will be returned. After a successful call on Start_Search, the object Search may have entries available, but it may have no entries available if

no files or directories match Pattern and Filter. The exception Name_Error is propagated if the string given by Directory does not identify an existing directory, or if Pattern does not allow the identification of any possible external file or directory. The exception Use_Error is propagated if the external environment does not support the searching of the directory with the given name (in the absence of Name_Error). When Start_Search propagates Name_Error or Use_Error, the object Search will have no entries available.

Implementation defined: The interpretation of a nonnull search pattern in Directories. 104.a/2

```
procedure End_Search (Search : in out Search_Type);
```
105/2

Ends the search represented by Search. After a successful call on End_Search, the object Search will have no entries available. 106/2

Ramification: The only way that a call to End_Search could be unsuccessful if Device_Error (see A.13) is raised because of an underlying failure (or bug). 106.a/2

```
function More_Entries (Search : in Search_Type) return Boolean;
```
107/2

Returns True if more entries are available to be returned by a call to Get_Next_Entry for the specified search object, and False otherwise. 108/2

```
procedure Get_Next_Entry (Search : in out Search_Type;
                          Directory_Entry : out Directory_Entry_Type);
```
109/2

{*AI05-0262-1*} Returns the next Directory_Entry for the search described by Search that matches the pattern and filter. If no further matches are available, Status_Error is raised. It is implementation defined as to whether the results returned by this subprogram are altered if the contents of the directory are altered while the Search object is valid (for example, by another program). The exception Use_Error is propagated if the external environment does not support continued searching of the directory represented by Search. 110/3

Implementation defined: The results of a Directories search if the contents of the directory are altered while a search is in progress. 110.a/2

```
{AI12-0286-1} procedure Search (
   Directory : in String;
   Pattern   : in String;
   Filter    : in Filter_Type := (others => True);
   Process   : not null access procedure (
       Directory_Entry : in Directory_Entry_Type))
   with Allows_Exit;
```
111/5

{*AI05-0092-1*} {*AI05-0262-1*} Searches in the directory named by Directory for entries matching Pattern and Filter. The subprogram designated by Process is called with each matching entry in turn. Pattern represents a pattern for matching file names. If Pattern is the null string, all items in the directory are matched; otherwise, the interpretation of Pattern is implementation defined. Only items that match Filter will be returned. The exception Name_Error is propagated if the string given by Directory does not identify an existing directory, or if Pattern does not allow the identification of any possible external file or directory. The exception Use_Error is propagated if the external environment does not support the searching of the directory with the given name (in the absence of Name_Error). 112/3

Discussion: "In turn" means that the calls to the subprogram designated by Process are not made in parallel; they can be made in any order but must be in sequence. 112.a/2

```
function Simple_Name (Directory_Entry : in Directory_Entry_Type)
   return String;
```
113/2

Returns the simple external name of the external file (including directories) represented by Directory_Entry. The format of the name returned is implementation defined. The exception Status_Error is propagated if Directory_Entry is invalid. 114/2

115/2
```
function Full_Name (Directory_Entry : in Directory_Entry_Type)
     return String;
```

116/2 Returns the full external name of the external file (including directories) represented by Directory_Entry. The format of the name returned is implementation defined. The exception Status_Error is propagated if Directory_Entry is invalid.

117/2
```
function Kind (Directory_Entry : in Directory_Entry_Type)
     return File_Kind;
```

118/2 Returns the kind of external file represented by Directory_Entry. The exception Status_Error is propagated if Directory_Entry is invalid.

119/2
```
function Size (Directory_Entry : in Directory_Entry_Type)
     return File_Size;
```

120/2 Returns the size of the external file represented by Directory_Entry. The size of an external file is the number of stream elements contained in the file. If the external file represented by Directory_Entry is not an ordinary file, the result is implementation defined. The exception Status_Error is propagated if Directory_Entry is invalid. The exception Constraint_Error is propagated if the file size is not a value of type File_Size.

121/2
```
function Modification_Time (Directory_Entry : in Directory_Entry_Type)
     return Ada.Calendar.Time;
```

122/2 Returns the time that the external file represented by Directory_Entry was most recently modified. If the external file represented by Directory_Entry is not an ordinary file, the result is implementation defined. The exception Status_Error is propagated if Directory_Entry is invalid. The exception Use_Error is propagated if the external environment does not support reading the modification time of the file represented by Directory_Entry.

Implementation Requirements

123/2 For Copy_File, if Source_Name identifies an existing external ordinary file created by a predefined Ada input-output package, and Target_Name and Form can be used in the Create operation of that input-output package with mode Out_File without raising an exception, then Copy_File shall not propagate Use_Error.

123.a/2 **Discussion:** This means that Copy_File will copy any file that the Ada programmer could copy (by writing some possibly complicated Ada code).

Implementation Advice

124/2 If other information about a file (such as the owner or creation date) is available in a directory entry, the implementation should provide functions in a child package Directories.Information to retrieve it.

124.a/2 **Implementation Advice:** Package Directories.Information should be provided to retrieve other information about a file.

124.b/2 **Implementation Note:** For Windows®, Directories.Information should contain at least the following routines:

124.c/5
```
{AI12-0302-1} package Ada.Directories.Information
   with Global => in out synchronized is
     -- System-specific directory information.
     -- Version for the Microsoft® Windows® operating system.
```

124.d/2
```
     function Creation_Time (Name : in String) return Ada.Calendar.Time;
```

124.e/2
```
     function Last_Access_Time (Name : in String) return Ada.Calendar.Time;
```

124.f/2
```
     function Is_Read_Only (Name : in String) return Boolean;
```

124.g/2
```
     function Needs_Archiving (Name : in String) return Boolean;
          -- This generally means that the file needs to be backed up.
          -- The flag is only cleared by backup programs.
```

124.h/2
```
     function Is_Compressed (Name : in String) return Boolean;
```

```
      function Is_Encrypted (Name : in String) return Boolean;                                    124.i/2
      function Is_Hidden (Name : in String) return Boolean;                                       124.j/2
      function Is_System (Name : in String) return Boolean;                                       124.k/2
      function Is_Offline (Name : in String) return Boolean;                                      124.l/2
      function Is_Temporary (Name : in String) return Boolean;                                    124.m/2
      function Is_Sparse (Name : in String) return Boolean;                                       124.n/2
      function Is_Not_Indexed (Name : in String) return Boolean;                                  124.o/2
      function Creation_Time (Directory_Entry : in Directory_Entry_Type)                          124.p/2
           return Ada.Calendar.Time;
      function Last_Access_Time (Directory_Entry : in Directory_Entry_Type)                       124.q/2
           return Ada.Calendar.Time;
      function Is_Read_Only (Directory_Entry : in Directory_Entry_Type) return Boolean;           124.r/2
      function Needs_Archiving (Directory_Entry : in Directory_Entry_Type) return                 124.s/2
Boolean;
          -- This generally means that the file needs to be backed up.
          -- The flag is only cleared by backup programs.
      function Is_Compressed (Directory_Entry : in Directory_Entry_Type) return Boolean;           124.t/2
      function Is_Encrypted (Directory_Entry : in Directory_Entry_Type) return Boolean;           124.u/2
      function Is_Hidden (Directory_Entry : in Directory_Entry_Type) return Boolean;              124.v/2
      function Is_System (Directory_Entry : in Directory_Entry_Type) return Boolean;              124.w/2
      function Is_Offline (Directory_Entry : in Directory_Entry_Type) return Boolean;             124.x/2
      function Is_Temporary (Directory_Entry : in Directory_Entry_Type) return Boolean;           124.y/2
      function Is_Sparse (Directory_Entry : in Directory_Entry_Type) return Boolean;              124.z/2
      function Is_Not_Indexed (Directory_Entry : in Directory_Entry_Type) return                  124.aa/2
Boolean;
      -- Additional implementation-defined subprograms allowed here.                             124.bb/2
end Ada.Directories.Information;
```

For Unix-like systems (Unix, POSIX, Linux, etc.), Directories.Information should contain at least the following routines: 124.cc/2

```
{AI12-0302-1} package Ada.Directories.Information                                                 124.dd/5
   with Global => in out synchronized is
   -- System-specific directory information.
   -- Unix and similar systems version.
      function Last_Access_Time (Name : in String) return Ada.Calendar.Time;                      124.ee/2
      function Last_Status_Change_Time (Name : in String) return Ada.Calendar.Time;               124.ff/2
      type Permission is                                                                          124.gg/2
        (Others_Execute, Others_Write, Others_Read,
         Group_Execute,  Group_Write,  Group_Read,
         Owner_Execute,  Owner_Write,  Owner_Read,
         Set_Group_ID,   Set_User_ID);
      type Permission_Set_Type is array (Permission) of Boolean;                                  124.hh/2
      function Permission_Set (Name : in String) return Permission_Set_Type;                      124.ii/2
      function Owner (Name : in String) return String;                                            124.jj/2
          -- Returns the image of the User_Id. If a definition of User_Id
          -- is available, an implementation-defined version of Owner
          -- returning User_Id should also be defined.
{AI05-0005-1}     function Group (Name : in String) return String;                              124.kk/3
          -- Returns the image of the Group_Id. If a definition of Group_Id
          -- is available, an implementation-defined version of Group
          -- returning Group_Id should also be defined.
      function Is_Block_Special_File (Name : in String) return Boolean;                           124.ll/2
      function Is_Character_Special_File (Name : in String) return Boolean;                       124.mm/2
      function Is_FIFO (Name : in String) return Boolean;                                         124.nn/2
      function Is_Symbolic_Link (Name : in String) return Boolean;                                124.oo/2
      function Is_Socket (Name : in String) return Boolean;                                       124.pp/2
      function Last_Access_Time (Directory_Entry : in Directory_Entry_Type)                       124.qq/2
         return Ada.Calendar.Time;
      function Last_Status_Change_Time (Directory_Entry : in Directory_Entry_Type)                124.rr/2
         return Ada.Calendar.Time;
```

124.ss/2
```
      function Permission_Set (Directory_Entry : in Directory_Entry_Type)
         return Permission_Set_Type;
```
124.tt/2
```
      function Owner (Directory_Entry : in Directory_Entry_Type) return String;
         -- See Owner above.
```
124.uu/2
```
      function Group (Directory_Entry : in Directory_Entry_Type) return String;
         -- See Group above.
```
124.vv/2
```
      function Is_Block_Special_File (Directory_Entry : in Directory_Entry_Type)
         return Boolean;
```
124.ww/2
```
      function Is_Character_Special_File (Directory_Entry : in Directory_Entry_Type)
         return Boolean;
```
124.xx/2
```
      function Is_FIFO (Directory_Entry : in Directory_Entry_Type) return Boolean;
```
124.yy/2
```
      function Is_Symbolic_Link (Directory_Entry : in Directory_Entry_Type)
         return Boolean;
```
124.zz/2
```
      function Is_Socket (Directory_Entry : in Directory_Entry_Type) return Boolean;
```
124.aaa/2
```
      -- Additional implementation-defined subprograms allowed here.
   end Ada.Directories.Information;
```

124.bbb/2 We give these definitions to give guidance so that every implementation for a given target is not unnecessarily different. Implementers are encouraged to make packages for other targets as similar to these as possible.

125/5 {*AI05-0231-1*} {*AI12-0439-1*} Start_Search and Search should raise Name_Error if Pattern is malformed, but not if it can represent a file in the directory but does not actually do so.

125.a/3 **Implementation Advice:** Directories.Start_Search and Directories.Search should raise Name_Error for malformed patterns.

126/2 Rename should be supported at least when both New_Name and Old_Name are simple names and New_Name does not identify an existing external file.

126.a/2 **Implementation Advice:** Directories.Rename should be supported at least when both New_Name and Old_Name are simple names and New_Name does not identify an existing external file.

126.b/2 **Discussion:** "Supported" includes raising an exception if either name is malformed, the file to rename doesn't exist, insufficient permission for the operation exists, or similar problems. But this advice requires implementations to document what they do, and tells implementers that simply raising Use_Error isn't acceptable.

127/5 NOTE 1 {*AI12-0442-1*} The operations Containing_Directory, Full_Name, Simple_Name, Base_Name, Extension, and Compose operate on file names, not external files. The files identified by these operations do not necessarily exist. Name_Error is raised only if the file name is malformed and cannot possibly identify a file. Of these operations, only the result of Full_Name depends on the current default directory; the result of the others depends only on their parameters.

128/2 NOTE 2 Using access types, values of Search_Type and Directory_Entry_Type can be saved and queried later. However, another task or application can modify or delete the file represented by a Directory_Entry_Type value or the directory represented by a Search_Type value; such a value can only give the information valid at the time it is created. Therefore, long-term storage of these values is not recommended.

129/2 NOTE 3 If the target system does not support directories inside of directories, then Kind will never return Directory and Containing_Directory will always raise Use_Error.

130/2 NOTE 4 If the target system does not support creation or deletion of directories, then Create_Directory, Create_Path, Delete_Directory, and Delete_Tree will always propagate Use_Error.

131/5 NOTE 5 {*AI12-0442-1*} {*AI12-0447-1*} To move a file or directory to a different location, use Rename. Most target systems will allow renaming of files from one directory to another. If the target file or directory can already exist, it should be deleted first.

131.a/2 **Discussion:** While Rename is only guaranteed to work for name changes within a single directory, its unlikely that implementers would purposely prevent functionality present in the underlying system from working. To move a file totally portably, it's necessary to handle failure of the Rename and fall back to Copy_File and Delete:

131.b
```
begin
   Rename (Source, Target);
exception
   when Use_Error =>
      Copy_File (Source, Target);
      Delete (Source);
end;
```

Extensions to Ada 95

131.c/2 {*AI95-00248-01*} Package Ada.Directories is new.

Inconsistencies With Ada 2005

{*AI05-0231-1*} **Correction**: Clarified when and which exceptions are raised for Start_Search, Search, Delete_Directory, and Rename. If an implementation followed the original incorrect wording, it might raise Use_Error instead of Name_Error for Start_Search and Search, Name_Error instead of Use_Error for Rename, and might have deleted a nonempty directory instead of raising Use_Error for Delete_Directory. The first two cases are very unlikely to matter in practice, and it unlikely that an implementation would have followed the latter implementation strategy, as it would be more work and would make Delete_Directory identical to Delete_Tree (which is obvious nonsense). 131.d/3

Incompatibilities With Ada 2005

{*AI05-0049-1*} A new enumeration type Name_Case_Kind and a new function Name_Case_Equivalence is added to Directories. If Directories is referenced in a use_clause, and an entity *E* with a defining_identifier of one of the new entities is defined in a package that is also referenced in a use_clause, the entity *E* may no longer be use-visible, resulting in errors. This should be rare and is easily fixed if it does occur. 131.e/3

Wording Changes from Ada 2005

{*AI05-0271-1*} **Correction:** We now explicitly say that the behavior of Create_Path and Copy_File is unspecified when Use_Error is raised. Nothing has changed here, as the behavior was (implicitly) unspecified in the 2007 Amendment. 131.f/3

Wording Changes from Ada 2012

{*AI12-0337-1*} {*AI12-0433-1*} **Correction:** Clarified the meaning of Simple_Name in the case that the parameter is a root directory. This was not previously described. 131.g/5

A.16.1 The Package Directories.Hierarchical_File_Names

{*AI05-0049-1*} The library package Directories.Hierarchical_File_Names is an optional package providing operations for file name construction and decomposition for targets with hierarchical file naming. 1/3

Static Semantics

{*AI05-0049-1*} If provided, the library package Directories.Hierarchical_File_Names has the following declaration: 2/3

```
{AI12-0241-1} {AI12-0302-1} package Ada.Directories.Hierarchical_File_Names                    3/5
   with Nonblocking, Global => in out synchronized is

   function Is_Simple_Name (Name : in String) return Boolean;                                  4/3

   function Is_Root_Directory_Name (Name : in String) return Boolean;                          5/3

   function Is_Parent_Directory_Name (Name : in String) return Boolean;                        6/3

   function Is_Current_Directory_Name (Name : in String) return Boolean;                       7/3

   function Is_Full_Name (Name : in String) return Boolean;                                    8/3

   function Is_Relative_Name (Name : in String) return Boolean;                                9/3

   function Simple_Name (Name : in String) return String                                       10/3
      renames Ada.Directories.Simple_Name;

   function Containing_Directory (Name : in String) return String                              11/3
      renames Ada.Directories.Containing_Directory;

   function Initial_Directory (Name : in String) return String;                                12/3

   function Relative_Name (Name : in String) return String;                                    13/3

   function Compose (Directory      : in String := "";                                         14/3
                     Relative_Name  : in String;
                     Extension      : in String := "") return String;

end Ada.Directories.Hierarchical_File_Names;                                                   15/3
```

{*AI05-0049-1*} {*AI05-0269-1*} In addition to the operations provided in package Directories.Hierarchical_File_Names, the operations in package Directories can be used with hierarchical 16/3

file names. In particular, functions Full_Name, Base_Name, and Extension provide additional capabilities for hierarchical file names.

17/3 `function Is_Simple_Name (Name : in String) return Boolean;`

18/3 Returns True if Name is a simple name, and returns False otherwise.

18.a/5 **Ramification:** {*AI12-0337-1*} Root directories are considered simple names, so this function will return True if Name represents a root. Use Is_Root_Directory if it is necessary to distinguish roots and other simple names.

19/3 `function Is_Root_Directory_Name (Name : in String) return Boolean;`

20/3 Returns True if Name is syntactically a root (a directory that cannot be decomposed further), and returns False otherwise.

21/3 `function Is_Parent_Directory_Name (Name : in String) return Boolean;`

22/3 Returns True if Name can be used to indicate symbolically the parent directory of any directory, and returns False otherwise.

22.a/3 **Implementation Note:** Is_Parent_Directory_Name returns True if and only if Name is ".." for both Unix and Windows®.

23/3 `function Is_Current_Directory_Name (Name : in String) return Boolean;`

24/3 Returns True if Name can be used to indicate symbolically the directory itself for any directory, and returns False otherwise.

24.a/3 **Implementation Note:** Is_Current_Directory_Name returns True if and only if Name is "." for both Unix and Windows®.

25/3 `function Is_Full_Name (Name : in String) return Boolean;`

26/3 Returns True if the leftmost directory part of Name is a root, and returns False otherwise.

27/3 `function Is_Relative_Name (Name : in String) return Boolean;`

28/3 {*AI05-0049-1*} {*AI05-0269-1*} Returns True if Name allows the identification of an external file (including directories and special files) but is not a full name, and returns False otherwise.

28.a/5 **Ramification:** {*AI12-0337-1*} Relative names include simple names other than root directories as a special case. This function returns False if the syntax of the name is incorrect.

29/3 `function Initial_Directory (Name : in String) return String;`

30/3 {*AI05-0049-1*} {*AI05-0248-1*} Returns the leftmost directory part in Name. [That is, it returns a root directory name (for a full name), or one of a parent directory name, a current directory name, or a simple name (for a relative name).] The exception Name_Error is propagated if the string given as Name does not allow the identification of an external file (including directories and special files).

31/3 `function Relative_Name (Name : in String) return String;`

32/3 Returns the entire file name except the Initial_Directory portion. The exception Name_Error is propagated if the string given as Name does not allow the identification of an external file (including directories and special files), or if Name has a single part (this includes if any of Is_Simple_Name, Is_Root_Directory_Name, Is_Parent_Directory_Name, or Is_Current_Directory_Name are True).

32.a/3 **Ramification:** The result might be a simple name.

33/3
```
function Compose (Directory      : in String := "";
                  Relative_Name  : in String;
                  Extension      : in String := "") return String;
```

34/3 Returns the name of the external file with the specified Directory, Relative_Name, and Extension. The exception Name_Error is propagated if the string given as Directory is not the null string and does not allow the identification of a directory, or if Is_Relative_Name (Relative_Name) is False,

or if the string given as Extension is not the null string and is not a possible extension, or if Extension is not the null string and Simple_Name (Relative_Name) is not a base name.

The result of Compose is a full name if Is_Full_Name (Directory) is True; result is a relative name otherwise. 35/3

Ramification: Name_Error is raised by Compose if Directory is not the null string, and both Is_Full_Name and Is_Relative_Name return False. 35.a/3

Discussion: A common security problem is to include a parent directory name in the middle of a file name; this is often used to navigate outside of an intended root directory. We considered attempting to prevent that case by having Compose detect it and raise an exception. But the extra rules necessary were more confusing than helpful. 35.b/3

{*AI12-0337-1*} We can say more about the details of these operations by adopting the notation of a subscript to specify how many path fragments a particular result has. Then, we can abbreviate "Full Name" as "Full" and "Relative Name" as "Rel". In this notation, Unix file name "a/b" is a Rel(2), "../c/d" is a Rel(3), and "/a/b" is a Full(2). Rel(1) is equivalent to a simple name that is not a root; thus we don't have to describe that separately. 35.c/5

In this notation, 35.d/3

35.e/3

```
For N>1,
Containing_Directory(Rel(N)) = Leftmost Rel(N-1),
Containing_Directory(Full(N)) = Leftmost Full(N-1),
Else if N = 1, raise Use_Error.
```

{*AI12-0337-1*} Similarly, 35.f/3

```
For N>1,                                                          35.g/3
Relative_Name(Rel(N)) = Rightmost Rel(N-1),
Relative_Name(Full(N)) = Rightmost Full(N-1),
Else if N = 1, raise Name_Error.
```

Finally, for Compose (ignoring the extension here): 35.h/3

```
Compose (Directory => Full(N), Relative_Name => Rel(M)) => Full(N+M)   35.i/3
Compose (Directory => Rel(N), Relative_Name => Rel(M)) => Rel(N+M)
Name_Error if Relative_Name is a Full(M).
```

We didn't try to write wording to reflect these details of these functions. 35.j/3

Implementation Advice

{*AI05-0049-1*} Directories.Hierarchical_File_Names should be provided for systems with hierarchical file naming, and should not be provided on other systems. 36/3

Implementation Advice: Directories.Hierarchical_File_Names should be provided for systems with hierarchical file naming, and should not be provided on other systems. 36.a/3

Implementation Note: This package should be provided when targeting Microsoft® Windows®, Unix, Linux, and most Unix-like systems. 36.b/3

NOTE 1 {*AI05-0049-1*} {*AI12-0442-1*} These operations operate on file names, not external files. The files identified by these operations do not necessarily exist. Name_Error is raised only as specified or if the file name is malformed and cannot possibly identify a file. The result of these operations depends only on their parameters. 37/5

NOTE 2 {*AI05-0049-1*} Containing_Directory raises Use_Error if Name does not have a containing directory, including when any of Is_Simple_Name, Is_Root_Directory_Name, Is_Parent_Directory_Name, or Is_Current_Directory_Name are True. 38/3

Ramification: In particular, the default directory is not used to find the containing directory either when Is_Parent_Directory_Name or Is_Current_Directory_Name is True. As noted above, these functions operate purely on the syntax of the file names and do not attempt to interpret them. If interpretation is needed, Directories.Full_Name can be to expand any shorthands used before calling Containing_Directory. 38.a/3

Extensions to Ada 2005

{*AI05-0049-1*} Package Ada.Directories.Hierarchical_File_Names is new. 38.b/3

A.16.2 The Packages Wide_Directories and Wide_Wide_Directories

1/5 {*AI12-0021-1*} The packages Wide_Directories and Wide_Wide_Directories provide operations for manipulating files and directories, and their names.

Static Semantics

2/5 {*AI12-0021-1*} The specification of package Wide_Directories is the same as for Directories (including its optional child packages Information and Hierarchical_File_Names), except that each occurrence of String is replaced by Wide_String.

3/5 {*AI12-0021-1*} The specification of package Wide_Wide_Directories is the same as for Directories (including its optional child packages Information and Hierarchical_File_Names), except that each occurrence of String is replaced by Wide_Wide_String.

Extensions to Ada 2012

3.a/5 {*AI12-0021-1*} These six packages are new.

A.17 The Package Environment_Variables

1/2 {*AI95-00370-01*} The package Environment_Variables allows a program to read or modify environment variables. Environment variables are name-value pairs, where both the name and value are strings. The definition of what constitutes an *environment variable*, and the meaning of the name and value, are implementation defined.

1.a/2 **Implementation defined:** The definition and meaning of an environment variable.

Static Semantics

2/2 {*AI95-00370-01*} The library package Environment_Variables has the following declaration:

3/5
```
{AI12-0241-1} {AI12-0302-1} package Ada.Environment_Variables
   with Preelaborate, Nonblocking, Global => in out synchronized is
```

4/2
```
   function Value (Name : in String) return String;
```

4.1/3
```
{AI05-0285-1}    function Value (Name : in String; Default : in String) return
String;
```

5/2
```
   function Exists (Name : in String) return Boolean;
```

6/2
```
   procedure Set (Name : in String; Value : in String);
```

7/2
```
   procedure Clear (Name : in String);
   procedure Clear;
```

8/5
```
{AI05-0248-1} {AI12-0286-1}    procedure Iterate
      (Process : not null access procedure (Name, Value : in String))
      with Allows_Exit;
```

9/2
```
end Ada.Environment_Variables;
```

10/2
```
function Value (Name : in String) return String;
```

11/2 {*AI95-00370-01*} If the external execution environment supports environment variables, then Value returns the value of the environment variable with the given name. If no environment variable with the given name exists, then Constraint_Error is propagated. If the execution environment does not support environment variables, then Program_Error is propagated.

```
function Value (Name : in String; Default : in String) return String;
```
11.1/3

{*AI05-0285-1*} If the external execution environment supports environment variables and an environment variable with the given name currently exists, then Value returns its value; otherwise, it returns Default. 11.2/3

```
function Exists (Name : in String) return Boolean;
```
12/2

{*AI95-00370-01*} {*AI05-0264-1*} If the external execution environment supports environment variables and an environment variable with the given name currently exists, then Exists returns True; otherwise, it returns False. 13/3

```
procedure Set (Name : in String; Value : in String);
```
14/2

{*AI95-00370-01*} {*AI05-0264-1*} If the external execution environment supports environment variables, then Set first clears any existing environment variable with the given name, and then defines a single new environment variable with the given name and value. Otherwise, Program_Error is propagated. 15/3

If implementation-defined circumstances prohibit the definition of an environment variable with the given name and value, then Constraint_Error is propagated. 16/2

Implementation defined: The circumstances where an environment variable cannot be defined. 16.a/2

It is implementation defined whether there exist values for which the call Set(Name, Value) has the same effect as Clear (Name). 17/2

Implementation defined: Environment names for which Set has the effect of Clear. 17.a/2

```
procedure Clear (Name : in String);
```
18/2

{*AI95-00370-01*} {*AI05-0264-1*} {*AI05-0269-1*} If the external execution environment supports environment variables, then Clear deletes all existing environment variables with the given name. Otherwise, Program_Error is propagated. 19/3

```
procedure Clear;
```
20/2

{*AI95-00370-01*} {*AI05-0264-1*} If the external execution environment supports environment variables, then Clear deletes all existing environment variables. Otherwise, Program_Error is propagated. 21/3

```
{AI05-0248-1} {AI12-0286-1} procedure Iterate
   (Process : not null access procedure (Name, Value : in String))
      with Allows_Exit;
```
22/5

{*AI95-00370-01*} {*AI05-0264-1*} If the external execution environment supports environment variables, then Iterate calls the subprogram designated by Process for each existing environment variable, passing the name and value of that environment variable. Otherwise, Program_Error is propagated. 23/3

If several environment variables exist that have the same name, Process is called once for each such variable. 24/2

Bounded (Run-Time) Errors

{*AI95-00370-01*} It is a bounded error to call Value if more than one environment variable exists with the given name; the possible outcomes are that: 25/2

- one of the values is returned, and that same value is returned in subsequent calls in the absence of changes to the environment; or 26/2
- Program_Error is propagated. 27/2

Erroneous Execution

28/2 {*AI95-00370-01*} Making calls to the procedures Set or Clear concurrently with calls to any subprogram of package Environment_Variables, or to any instantiation of Iterate, results in erroneous execution.

29/2 Making calls to the procedures Set or Clear in the actual subprogram corresponding to the Process parameter of Iterate results in erroneous execution.

Documentation Requirements

30/2 {*AI95-00370-01*} An implementation shall document how the operations of this package behave if environment variables are changed by external mechanisms (for instance, calling operating system services).

30.a/2 **Documentation Requirement:** The behavior of package Environment_Variables when environment variables are changed by external mechanisms.

Implementation Permissions

31/2 {*AI95-00370-01*} An implementation running on a system that does not support environment variables is permitted to define the operations of package Environment_Variables with the semantics corresponding to the case where the external execution environment does support environment variables. In this case, it shall provide a mechanism to initialize a nonempty set of environment variables prior to the execution of a partition.

Implementation Advice

32/2 {*AI95-00370-01*} If the execution environment supports subprocesses, the currently defined environment variables should be used to initialize the environment variables of a subprocess.

32.a/2 **Implementation Advice:** If the execution environment supports subprocesses, the current environment variables should be used to initialize the environment variables of a subprocess.

33/2 Changes to the environment variables made outside the control of this package should be reflected immediately in the effect of the operations of this package. Changes to the environment variables made using this package should be reflected immediately in the external execution environment. This package should not perform any buffering of the environment variables.

33.a/2 **Implementation Advice:** Changes to the environment variables made outside the control of Environment_Variables should be reflected immediately.

Extensions to Ada 95

33.b/2 {*AI95-00370-01*} Package Environment_Variables is new.

Incompatibilities With Ada 2005

33.c/3 {*AI05-0285-1*} A new overloaded function Value is added to Environment_Variables. If Environment_Variables is referenced in a use_clause, and an entity *E* with the name Value is defined in a package that is also referenced in a use_clause, the entity *E* may no longer be use-visible, resulting in errors. This should be rare and is easily fixed if it does occur.

A.17.1 The Packages Wide_Environment_Variables and Wide_Wide_Environment_Variables

1/5 {*AI12-0021-1*} The packages Wide_Environment_Variables and Wide_Wide_Environment_Variables allow a program to read or modify environment variables.

Static Semantics

{*AI12-0021-1*} The specification of package Wide_Environment_Variables is the same as for Environment_Variables, except that each occurrence of String is replaced by Wide_String. 2/5

{*AI12-0021-1*} The specification of package Wide_Wide_Environment_Variables is the same as for Environment_Variables, except that each occurrence of String is replaced by Wide_Wide_String. 3/5

Extensions to Ada 2012

{*AI12-0021-1*} These packages are new. 3.a/5

A.18 Containers

{*AI95-00302-03*} This clause presents the specifications of the package Containers and several child packages, which provide facilities for storing collections of elements. 1/2

Term entry: container — structured object that represents a collection of elements all of the same (potentially class-wide) type, such as a vector or a tree 1.a/5
Note: Several predefined container types are provided by the children of package Ada.Containers (see A.18.1).

{*AI95-00302-03*} {*AI12-0196-1*} {*AI12-0416-1*} A variety of sequence and associative containers are provided. Each container package defines a *cursor* type as well as a container type. A cursor is a reference to an element within a container. Many operations on cursors are common to all of the containers. A cursor referencing an element in a container is considered to be overlapping only with the element itself. 2/5

Reason: {*AI12-0005-1*} {*AI12-0196-1*} The last sentence is intended to clarify that operations that just use a cursor do not interfere if the cursor objects designated different elements of the container in terms of the concurrent call rules of Annex A. 2.a/5

Ramification: {*AI12-0196-1*} A cursor is not considered to overlap with other elements of the associated container, thus parallel operations involving a set of cursors each operating on mutually exclusive sets of elements from the same container are expected to work. 2.b/5

Discussion: {*AI12-0416-1*} We use the term “container” alone when it is clear from context what kind of entity (package, type, or object) that we are talking about. Otherwise, we use “container package”, “container type”, or “container object”. Note that "container type" is defined in 4.3.5 for a different usage; in all of A.18 we mean “container type” to be one of the primary types declared in the child packages of package Containers, such as Vector, List, or Map. 2.c/5

{*AI12-0111-1*} {*AI12-0439-1*} Some operations of the language-defined child units of Ada.Containers have access-to-subprogram parameters. To ensure such operations are well-defined, they guard against certain actions by the designated subprogram. An action on a container that can add or remove an element is considered to *tamper with cursors*, and these are prohibited during all such operations. An action on a container that can replace an element with one of a different size is considered to *tamper with elements*, and these are prohibited during certain of such operations. The details of the specific actions that are considered to tamper with cursors or elements are defined for each child unit of Ada.Containers. 2.1/5

{*AI12-0111-1*} Several of the language-defined child units of Ada.Containers include a nested package named Stable, which provides a view of a container that prohibits any operations that would tamper with elements. By using a Stable view for manipulating a container, the number of tampering checks performed while performing the operations can be reduced. The details of the Stable subpackage are defined separately for each child unit of Ada.Containers that includes such a nested package. 2.2/5

{*AI95-00302-03*} Within this clause we provide Implementation Advice for the desired average or worst case time complexity of certain operations on a container. This advice is expressed using the Landau symbol *O*(X). Presuming f is some function of a length parameter N and t(N) is the time the operation takes (on average or worst case, as specified) for the length N, a complexity of *O*(f(N)) means that there exists a finite A such that for any N, t(N)/f(N) < A. 3/2

3.a/2 **Discussion:** Of course, an implementation can do better than a specified *O*(f(N)): for example, *O*(1) meets the requirements for *O*(log N).

3.b/2 This concept seems to have as many names as there are authors. We used “Landau symbol” because that's what our reference does. But we'd also seen this referred as big-O notation (sometimes written as *big-oh*), and as Bachmann notation. Whatever the name, it always has the above definition.

4/2 If the advice suggests that the complexity should be less than *O*(f(N)), then for any arbitrarily small positive real D, there should exist a positive integer M such that for all N > M, t(N)/f(N) < D.

5/5 {*AI05-0001-1*} {*AI05-0044-1*} {*AI12-0416-1*} When a formal function is used to provide an ordering for a container, it is generally required to define a strict weak ordering. A function "<" defines a *strict weak ordering* if it is irreflexive, asymmetric, transitive, and in addition, if $x < y$ for any values x and y, then for all other values z, ($x < z$) or ($z < y$). Elements are in a *smallest first* order using such an operator if, for every element y with a predecessor x in the order, ($y < x$) is false.

5.a.1/5 **Reason:** {*AI12-0416-1*} Given a "<" operator that provides a strict weak ordering, knowing that ($y < x$) is false is enough to know that ($x <= y$) is true. For a strict weak ordering, ($x = y$) when both ($x < y$) and ($y < x$) are false. Therefore, it is not necessary to use the "=" operator or test ($x < y$). We only need to discuss adjacent elements since a strict weak ordering is transitive.

Language Design Principles

5.a/3 {*AI95-00302-03*} {*AI05-0299-1*} This subclause provides a number of useful containers for Ada. Only the most useful containers are provided. Ones that are relatively easy to code, redundant, or rarely used are omitted from this set, even if they are generally included in containers libraries.

5.b/2 The containers packages are modeled on the Standard Template Library (STL), an algorithms and data structure library popularized by Alexander Stepanov, and included in the C++ standard library. The structure and terminology differ from the STL where that better maps to common Ada usage. For instance, what the STL calls “iterators” are called “cursors” here.

5.c/2 The following major nonlimited containers are provided:

5.d/2 - (Expandable) Vectors of any nonlimited type;

5.e/2 - Doubly-linked Lists of any nonlimited type;

5.f/2 - Hashed Maps keyed by any nonlimited hashable type, and containing any nonlimited type;

5.g/2 - Ordered Maps keyed by any nonlimited ordered type, and containing any nonlimited type;

5.h/3 - {*AI05-0136-1*} Hashed Sets of any nonlimited hashable type;

5.i/3 - {*AI05-0136-1*} Ordered Sets of any nonlimited ordered type;

5.i.1/3 - {*AI05-0136-1*} Multiway Trees of any nonlimited type;

5.i.2/3 - {*AI05-0069-1*} Holders of any (indefinite) nonlimited type;

5.i.3/3 - {*AI05-0159-1*} Synchronized queues of any definite nonlimited type; and

5.i.4/3 - {*AI05-0159-1*} Priority queues of any definite nonlimited type.

5.j/3 {*AI05-0001-1*} Separate versions for definite and indefinite element types are provided, as those for definite types can be implemented more efficiently. Similarly, a separate bounded version is provided in order to give more predictable memory usage.

5.k/2 Each container includes a cursor, which is a reference to an element within a container. Cursors generally remain valid as long as the container exists and the element referenced is not deleted. Many operations on cursors are common to all of the containers. This makes it possible to write generic algorithms that work on any kind of container.

5.l/2 The containers packages are structured so that additional packages can be added in the future. Indeed, we hope that these packages provide the basis for a more extensive secondary standard for containers.

5.m/2 If containers with similar functionality (but different performance characteristics) are provided (by the implementation or by a secondary standard), we suggest that a prefix be used to identify the class of the functionality: "Ada.Containers.Bounded_Sets" (for a set with a maximum number of elements); "Ada.Containers.Protected_Maps" (for a map which can be accessed by multiple tasks at one time); "Ada.Containers.Persistent_Vectors" (for a persistent vector which continues to exist between executions of a program) and so on.

5.n/2 Note that the language already includes several requirements that are important to the use of containers. These include:

5.o/5 - {*AI12-0196-1*} Library packages must allow concurrent calls – multiple tasks can use the packages as long as they operate on separate containers. Thus, it is only necessary for a user to protect a container if a single

container needs to be used by multiple tasks and concurrent calls to operations of the container have overlapping parameters.

- Language-defined types must stream "properly". That means that the stream attributes can be used to implement persistence of containers when necessary, and containers can be passed between partitions of a program. 5.p/2
- Equality of language-defined types must compose “properly”. This means that the version of "=" directly used by users is the same one that will be used in generics and in predefined equality operators of types with components of the containers and/or cursors. This prevents the abstraction from breaking unexpectedly. 5.q/2
- {*AI05-0048-1*} Redispatching is not allowed (unless it is required). That means that overriding a container operation will not change the behavior of any other predefined container operation. This provides a stable base for extensions. 5.q.1/3

{*AI12-0258-1*} If a container's element type is controlled, the point at which the element is finalized will depend on the implementation of the container. For certain kinds of containers, we require finalization behavior based on the canonical implementation of the container (see the Implementation Requirements below). For the "normal" containers, we do not specify precisely where this will happen (it will happen no later than the finalization of the container, of course) in order to give implementations flexibility to cache, block, split , or reusethe nodes of the container. 5.r/5

This paragraph was deleted.{*AI12-0258-1*} 5.s/5

The use of controlled types also brings up the possibility of failure of finalization (and thus deallocation) of an element. This is a “serious bug”, as AI95-179 puts it, so we don't try to specify what happens in that case. The implementation should propagate the exception. 5.t/2

Implementation Note: It is expected that exceptions propagated from these operations do not damage containers. That is, if Storage_Error is propagated because of an allocation failure, or Constraint_Error is propagated by the assignment of elements, the container can continue to be used without further exceptions. The intent is that it should be possible to recover from errors without losing data. We don't try to state this formally in most cases, because it is hard to define precisely what is and is not allowed behavior. 5.u/2

Implementation Note: {*AI12-0005-1*} When this clause says that the behavior of something is unspecified, we really mean that any result of executing Ada code short of erroneous execution is allowed. We do not mean that memory not belonging to the parameters of the operation can be trashed. When we mean to allow erroneous behavior, we specifically say that execution is erroneous. All this means that, if the containers are written in Ada, checks should not be suppressed or removed assuming some behavior of other code, and that the implementation should take care to avoid creating internal dangling accesses by assuming behavior from generic formals that can't be guaranteed. We don't try to say this normatively because it would be fairly complex, and implementers are unlikely to increase their support costs by fielding implementations that are unstable if given buggy hash functions, et al. 5.v/5

Static Semantics

{*AI12-0035-1*} {*AI12-0449-1*} Certain subprograms declared within instances of some of the generic packages presented in this clause are said to *perform indefinite insertion*. These subprograms are those corresponding (in the sense of the copying described in 12.3) to subprograms that have formal parameters of a generic formal indefinite type and that are identified as performing indefinite insertion in the subclause defining the generic package. 6/5

{*AI12-0035-1*} {*AI12-0449-1*} If a subprogram performs indefinite insertion, then certain run-time checks are performed as part of a call to the subprogram; if any of these checks fail, then the resulting exception is propagated to the caller and the container is not modified by the call. These checks are performed for each parameter corresponding (in the sense of the copying described in 12.3) to a parameter in the corresponding generic whose type is a generic formal indefinite type. The checks performed for a given parameter are those checks explicitly specified in 4.8 that would be performed as part of the evaluation of an initialized allocator whose access type is declared immediately within the instance, where: 7/5

- the value of the qualified_expression is that of the parameter; and 8/4
- the designated subtype of the access type is the subtype of the parameter; and 9/4
- finalization of the collection of the access type has started if and only if the finalization of the instance has started. 10/4

10.a/5 **Discussion:** {*AI12-0449-1*} The phrase "explicitly specified" means those checks for which 4.8 includes the phrase "<some exception> is raised if ...". It does not refer, for example, to any checks performed as part of any subtype conversion. In particular, this wording includes the checks described in 4.8 to be performed in the case of a class-wide designated type, and of a designated subtype that has access discriminant parts. These checks are needed to prevent containers from outliving their contained (Element_Type or Key_Type) values.

10.b/4 **Implementation Note:** These rules have a dual purpose. Mainly, we are *requiring* checks needed to prevent dangling references. As a side effect, we are also *allowing* checks needed to permit an implementation of a container generic to make use of access types in a straightforward way. As an example of the second purpose, suppose that an implementation does declare such an access type and suppose further that the finalization of the collection of the access type has started. These rules allow Program_Error to be propagated in this case (as specified in 4.8); this is necessary to allow an all-Ada implementation of these packages.

Implementation Requirements

11/5 {*AI12-0258-1*} For an indefinite container (one whose type is defined in an instance of a child package of Containers whose defining_identifier contains "Indefinite"), each element of the container shall be created when it is inserted into the container and finalized when it is deleted from the container (or when the container object is finalized if the element has not been deleted). For a bounded container (one whose type is defined in an instance of a child package of Containers whose defining_identifier starts with "Bounded") that is not an indefinite container, all of the elements of the capacity of the container shall be created and default initialized when the container object is created; the elements shall be finalized when the container object is finalized. [For other kinds of containers, when elements are created and finalized is unspecified.]

11.a/5 **Ramification:** This allows a user to be able to reason about the behavior of elements that have controlled parts. In most cases, such elements need to be stored in an indefinite container.

11.b/5 **Implementation Note:** If the containers are implemented in Ada, this implies that elements for an indefinite container are allocated individually, and that a bounded container contains an array of elements or other data structure that is initialized for the entire capacity of the container when it is created. There is no such restriction on the implementation of the "normal" containers; these can be handled in any way convenient to the implementation — in particular, node reuse is allowed.

12/5 {*AI12-0112-1*} For an instance *I* of a container package with a container type, the specific type *T* of the object returned from a function that returns an object of an iterator interface, as well as the primitive operations of *T*, shall be nonblocking. The Global aspect specified for *T* and the primitive operations of *T* shall be (**in all, out synchronized**) or a specification that allows access to fewer global objects.

12.a/5 **Implementation Note:** This requires that the traversal and iteration operations of a container do not create, destroy, or assign any objects of a formal type of *I*, nor call any formal subprograms of *I*. Those objects and subprograms might be blocking (depending on the actual parameters). We put similar requirements on the individual traversal operations in the container package definitions.

12.b/5 **Reason:** These requirements allows users to use container iterators inside of parallel constructs, regardless of the actual parameters to the instantiation. If such an iterator allowed blocking, it would be illegal inside of a parallel construct (see 9.5). If such an iterator allowed writing of unsynchronized global objects, it would be illegal when the default conflict checking policy is in effect (see 9.10.1). These requirements include sequential iterators; the iterator does not need to appear in a parallel loop to trigger these requirements.

12.c/5 **Discussion:** We have to give these requirements as a text rule, as there is no place to declare suitable aspects. The specific type of a container iterator is declared by the implementation and is not part of the visible specification (iterator functions just return a value of a class-wide type). The iterator interface itself cannot impose such a requirement since it needs to be able to work with user-defined types that do need to allow blocking. We give this as a global requirement to avoid duplication.

Extensions to Ada 95

12.d/3 {*AI95-00302-03*} {*AI05-0299-1*} This subclause is new. It just provides an introduction to the following subclauses.

Wording Changes from Ada 2005

12.e/3 {*AI05-0044-1*} **Correction:** Added a definition of strict weak ordering.

Extensions to Ada 2012

{*AI12-0196-1*} **Correction:** We now say that a cursor only overlaps with the element it designates, rather than with the whole container. This allows some reading operations to operate on the container in parallel without separate synchronization. 12.f/5

Wording Changes from Ada 2012

{*AI05-0035-1*} **Corrigendum:** Added a definition of "performs indefinite insertion". This is used in other subclauses and any resulting inconsistencies are documented there. 12.g/4

{*AI12-0111-1*} Moved the basic description of tampering checks here, to cut duplication in description of the individual containers. Added a description of stable views of containers. 12.h/5

{*AI12-0112-1*} Added a global requirement that iterators returned from containers are nonblocking if the instance is nonblocking. 12.i/5

{*AI12-0258-1*} **Correction:** Defined when objects are created and finalized for Bounded and Indefinite containers, so that these can be used reliably with controlled element types. This is not incompatible as this behavior was previously unspecified; code depending on specific behavior was wrong. 12.j/5

{*AI12-0005-1*} {*AI12-0416-1*} Added a definition of "smallest first" ordering, so that the behavior of the Sort procedures when elements are equal is well-defined. 12.k/5

A.18.1 The Package Containers

{*AI95-00302-03*} The package Containers is the root of the containers subsystem. 1/2

Static Semantics

{*AI95-00302-03*} The library package Containers has the following declaration: 2/2

```
{AI12-0414-1} package Ada.Containers                                    3/5
   with Pure is

   type Hash_Type is mod implementation-defined;                        4/2

   type Count_Type is range 0 .. implementation-defined;                5/2

{AI05-0001-1}    Capacity_Error : exception;                            5.1/3

end Ada.Containers;                                                     6/2
```

{*AI95-00302-03*} Hash_Type represents the range of the result of a hash function. Count_Type represents the (potential or actual) number of elements of a container. 7/2

Implementation defined: The value of Containers.Hash_Type'Modulus. The value of Containers.Count_Type'Last. 7.a/2

{*AI05-0262-1*} Capacity_Error is raised when the capacity of a container is exceeded. 7.1/3

Implementation Advice

{*AI95-00302-03*} Hash_Type'Modulus should be at least 2**32. Count_Type'Last should be at least 2**31–1. 8/2

Implementation Advice: Containers.Hash_Type'Modulus should be at least 2**32. Containers.Count_Type'Last should be at least 2**31–1. 8.a/2

Discussion: This is not a requirement so that these types can be declared properly on machines with native sizes that are not 32 bits. For instance, a 24-bit target could use 2**24 for Hash_Type'Modulus. 8.b/2

Extensions to Ada 95

{*AI95-00302-03*} The package Containers is new. 8.c/2

Incompatibilities With Ada 2005

{*AI05-0001-1*} Exception Capacity_Error is added to Containers. If Containers is referenced in a use_clause, and an entity with the name Capacity_Error is defined in a package that is also referenced in a use_clause, the entity Capacity_Error may no longer be use-visible, resulting in errors. This should be rare and is easily fixed if it does occur. 8.d/3

A.18.2 The Generic Package Containers.Vectors

1/2 The language-defined generic package Containers.Vectors provides private types Vector and Cursor, and a set of operations for each type. A vector container allows insertion and deletion at any position, but it is specifically optimized for insertion and deletion at the high end (the end with the higher index) of the container. A vector container also provides random access to its elements.

2/2 A vector container behaves conceptually as an array that expands as necessary as items are inserted. The *length* of a vector is the number of elements that the vector contains. The *capacity* of a vector is the maximum number of elements that can be inserted into the vector prior to it being automatically expanded.

3/2 Elements in a vector container can be referred to by an index value of a generic formal type. The first element of a vector always has its index value equal to the lower bound of the formal type.

4/2 A vector container may contain *empty elements*. Empty elements do not have a specified value.

4.a/2 **Implementation Note:** Vectors are not intended to be sparse (that is, there are elements at all defined positions). Users are expected to use other containers (like a Map) when they need sparse structures (there is a Note to this effect at the end of this subclause).

4.b/2 The internal array is a conceptual model of a vector. There is no requirement for an implementation to be a single contiguous array.

Static Semantics

5/2 {*AI95-00302-03*} The generic library package Containers.Vectors has the following declaration:

6/5
```
{AI05-0084-1} {AI05-0212-1} {AI12-0112-1} with Ada.Iterator_Interfaces;
generic
   type Index_Type is range <>;
   type Element_Type is private;
   with function "=" (Left, Right : Element_Type)
      return Boolean is <>;
package Ada.Containers.Vectors
   with Preelaborate, Remote_Types,
        Nonblocking, Global => in out synchronized is
```

6.a/5 **Discussion:** {*AI12-0112-1*} For the Global aspect, any side-effects of the actual parameters of an instance are ignored. So Global => **in out synchronized** means that the only global side-effects allowed are associated with the actual generic parameters of the instance or with any synchronized state. Unsynchronized package state is not allowed for any container package, and pure packages do not allow any package state at all (they typically have Global => **null**).

6.b/5 {*AI12-0112-1*} Similarly, when Nonblocking is set to True for a generic unit, it still includes the blocking effects of the actual parameters to the instance. Thus, the only blocking allowed is that associated with the actual generic parameters. If none of the actual paramerters allow blocking, then no operation of the generic instance may block.

7/2
```
   subtype Extended_Index is
      Index_Type'Base range
         Index_Type'First-1 ..
         Index_Type'Min (Index_Type'Base'Last - 1, Index_Type'Last) + 1;
   No_Index : constant Extended_Index := Extended_Index'First;
```

7.a/5 **Ramification:** {*AI12-0112-1*} The base type of a scalar type is always nonblocking and has Global => **null**. Therefore, so long as this type is used in the implementation, whether or not the actual type for Index_Type allows blocking or side-effects does not matter. Therefore, we require that operations that only operate on the container implementation be nonblocking and have Global => **null** regardless of the actual parameters.

```
{AI05-0212-1} {AI12-0111-1} {AI12-0112-1} {AI12-0212-1} {AI12-0339-1} {AI12-0399-1}    8/5
{AI12-0400-1}    type Vector is tagged private
      with Constant_Indexing => Constant_Reference,
           Variable_Indexing => Reference,
           Default_Iterator  => Iterate,
           Iterator_Element  => Element_Type,
           Iterator_View     => Stable.Vector,
           Aggregate         => (Empty          => Empty,
                                 Add_Unnamed    => Append,
                                 New_Indexed    => New_Vector,
                                 Assign_Indexed => Replace_Element),
           Stable_Properties => (Length, Capacity,
                                 Tampering_With_Cursors_Prohibited,
                                 Tampering_With_Elements_Prohibited),
           Default_Initial_Condition =>
              Length (Vector) = 0 and then
              (not Tampering_With_Cursors_Prohibited (Vector)) and then
              (not Tampering_With_Elements_Prohibited (Vector)),
           Preelaborable_Initialization;

{AI12-0399-1}    type Cursor is private                                                  9/5
      with Preelaborable_Initialization;

   Empty_Vector : constant Vector;                                                      10/2

   No_Element : constant Cursor;                                                        11/2

{AI05-0212-1} {AI12-0112-1}    function Has_Element (Position : Cursor) return       11.1/5
Boolean
      with Nonblocking, Global => in all, Use_Formal => null;
```

Discussion: {*AI12-0112-1*} Any operation that takes a cursor but no vector can read the vector associated with the cursor. We only know that there is some object of type Vector. Since we don't have a global specification that describes all objects of a specific type, we have to allow reading any object by specifying **in all**. For such functions, we don't allow writing any object, even those associated with generic formal parameters, thus we also specify Use_Formal => **null**. 11.a/5

```
{AI12-0112-1}    function Has_Element (Container : Vector; Position : Cursor)        11.2/5
      return Boolean
      with Nonblocking, Global => null, Use_Formal => null;
```

Discussion: {*AI12-0112-1*} For operations that do not depend on any of the operations of the generic formal parameters (including those of formal types), we specify that the operation has no side-effects of any kind. This requires specifying that there is no dependence on the generic formal parameters with Use_Formal => **null** in addition to no usual side-effects with **null**. We also specify Nonblocking on such operations in order that the operation never blocks even if some of the actual parameters allow blocking. 11.b/5

```
{AI05-0212-1}    package Vector_Iterator_Interfaces is new                           11.3/5
       Ada.Iterator_Interfaces (Cursor, Has_Element);

   function "=" (Left, Right : Vector) return Boolean;                              12/2

{AI12-0112-1}    function Tampering_With_Cursors_Prohibited                          12.1/5
      (Container : Vector) return Boolean
      with Nonblocking, Global => null, Use_Formal => null;

{AI12-0112-1}    function Tampering_With_Elements_Prohibited                         12.2/5
      (Container : Vector) return Boolean
      with Nonblocking, Global => null, Use_Formal => null;

{AI12-0112-1}    function Maximum_Length return Count_Type                           12.3/5
      with Nonblocking, Global => null, Use_Formal => null;

{AI12-0339-1}    function Empty (Capacity : Count_Type := implementation-defined)    12.4/5
      return Vector
      with Pre  => Capacity <= Maximum_Length
                     or else raise Constraint_Error,
           Post =>
              Capacity (Empty'Result) >= Capacity and then
              not Tampering_With_Elements_Prohibited (Empty'Result) and then
              not Tampering_With_Cursors_Prohibited (Empty'Result) and then
              Length (Empty'Result) = 0;
```

13/5 {*AI12-0112-1*}
```
function To_Vector (Length : Count_Type) return Vector
   with Pre  => Length <= Maximum_Length or else raise Constraint_Error,
        Post =>
           To_Vector'Result.Length = Length and then
           not Tampering_With_Elements_Prohibited (To_Vector'Result)
             and then
           not Tampering_With_Cursors_Prohibited (To_Vector'Result)
             and then
           To_Vector'Result.Capacity >= Length;
```

14/5 {*AI12-0112-1*}
```
function To_Vector
  (New_Item : Element_Type;
   Length   : Count_Type) return Vector
   with Pre  => Length <= Maximum_Length or else raise Constraint_Error,
        Post =>
           To_Vector'Result.Length = Length and then
           not Tampering_With_Elements_Prohibited (To_Vector'Result)
             and then
           not Tampering_With_Cursors_Prohibited (To_Vector'Result)
             and then
           To_Vector'Result.Capacity >= Length;
```

14.1/5 {*AI12-0212-1*}
```
function New_Vector (First, Last : Index_Type) return Vector is
   (To_Vector (Count_Type (Last - First + 1)))
with Pre => First = Index_Type'First;
```

15/5 {*AI12-0112-1*}
```
function "&" (Left, Right : Vector) return Vector
   with Pre  => Length (Left) <= Maximum_Length - Length (Right)
                  or else raise Constraint_Error,
        Post => Length (Vectors."&"'Result) =
                  Length (Left) + Length (Right) and then
                not Tampering_With_Elements_Prohibited
                  (Vectors."&"'Result) and then
                not Tampering_With_Cursors_Prohibited
                  (Vectors."&"'Result) and then
                Vectors."&"'Result.Capacity >=
                  Length (Left) + Length (Right);
```

16/5 {*AI12-0112-1*}
```
function "&" (Left  : Vector;
              Right : Element_Type) return Vector
   with Pre  => Length (Left) <= Maximum_Length - 1
                  or else raise Constraint_Error,
        Post => Vectors."&"'Result.Length = Length (Left) + 1 and then
                not Tampering_With_Elements_Prohibited
                  (Vectors."&"'Result) and then
                not Tampering_With_Cursors_Prohibited
                  (Vectors."&"'Result) and then
                Vectors."&"'Result.Capacity >= Length (Left) + 1;
```

17/5 {*AI12-0112-1*}
```
function "&" (Left  : Element_Type;
              Right : Vector) return Vector
   with Pre  => Length (Right) <= Maximum_Length - 1
                  or else raise Constraint_Error,
        Post => Length (Vectors."&"'Result) = Length (Right) + 1 and then
                not Tampering_With_Elements_Prohibited
                  (Vectors."&"'Result) and then
                not Tampering_With_Cursors_Prohibited
                  (Vectors."&"'Result) and then
                Vectors."&"'Result.Capacity >= Length (Right) + 1;
```

18/5 {*AI12-0112-1*}
```
function "&" (Left, Right  : Element_Type) return Vector
   with Pre  => Maximum_Length >= 2 or else raise Constraint_Error,
        Post => Length ("&"'Result) = 2 and then
                not Tampering_With_Elements_Prohibited
                  (Vectors."&"'Result) and then
                not Tampering_With_Cursors_Prohibited
                  (Vectors."&"'Result) and then
                Vectors."&"'Result.Capacity >= 2;
```

19/5 {*AI12-0112-1*}
```
function Capacity (Container : Vector) return Count_Type
   with Nonblocking, Global => null, Use_Formal => null;
```

```
{AI12-0112-1}    procedure Reserve_Capacity (Container : in out Vector;                20/5
                                             Capacity  : in     Count_Type)
      with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                      or else raise Program_Error,
           Post => Container.Capacity >= Capacity;

{AI12-0112-1}    function Length (Container : Vector) return Count_Type                21/5
      with Nonblocking, Global => null, Use_Formal => null;

{AI12-0112-1}    procedure Set_Length (Container : in out Vector;                      22/5
                                       Length    : in     Count_Type)
      with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                      or else raise Program_Error) and then
                    (Length <= Maximum_Length
                      or else raise Constraint_Error),
           Post => Container.Length = Length and then
                   Capacity (Container) >= Length;

{AI12-0112-1}    function Is_Empty (Container : Vector) return Boolean                 23/5
      with Nonblocking, Global => null, Use_Formal => null,
           Post => Is_Empty'Result = (Length (Container) = 0);

{AI12-0112-1}    procedure Clear (Container : in out Vector)                           24/5
      with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                     or else raise Program_Error,
           Post => Length (Container) = 0;

{AI12-0112-1}    function To_Cursor (Container : Vector;                               25/5
                                     Index     : Extended_Index) return Cursor
      with Post => (if Index in
                      First_Index (Container) .. Last_Index (Container)
                    then Has_Element (Container, To_Cursor'Result)
                    else To_Cursor'Result = No_Element),
           Nonblocking, Global => null, Use_Formal => null;

{AI12-0112-1}    function To_Index (Position  : Cursor) return Extended_Index          26/5
      with Nonblocking, Global => in all;

{AI12-0112-1}    function To_Index (Container : Vector;                                26.1/5
                                    Position  : Cursor) return Extended_Index
      with Pre  => Position = No_Element or else
                   Has_Element (Container, Position) or else
                      raise Program_Error,
           Post => (if Position = No_Element then To_Index'Result = No_Index
                    else To_Index'Result in First_Index (Container) ..
                                            Last_Index (Container)),
           Nonblocking, Global => null, Use_Formal => null;

{AI12-0112-1}    function Element (Container : Vector;                                 27/5
                                   Index     : Index_Type)
      return Element_Type
      with Pre  => Index in
                     First_Index (Container) .. Last_Index (Container)
                   or else raise Constraint_Error,
           Nonblocking, Global => null, Use_Formal => Element_Type;
```

Discussion: {*AI12-0112-1*} Here the Nonblocking and Global contracts are saying that Element depends on the properties of the actual for Element_Type, but not on the properties of the actuals for Index_Type or "=". This is necessary as copying the element may require calling Adjust and Finalize for the actual Element_Type, and those may have side-effects or block. 27.a/5

```
{AI12-0112-1}    function Element (Position : Cursor) return Element_Type              28/5
      with Pre  => Position /= No_Element or else raise Constraint_Error,
           Nonblocking, Global => in all, Use_Formal => Element_Type;

{AI12-0112-1}    function Element (Container : Vector;                                 28.1/5
                                   Position  : Cursor) return Element_Type
      with Pre => (Position /= No_Element or else
                     raise Constraint_Error) and then
                  (Has_Element (Container, Position)
                     or else raise Program_Error),
           Nonblocking, Global => null, Use_Formal => Element_Type;
```

29/5 {*AI12-0112-1*} **procedure** Replace_Element (Container : **in out** Vector;
Index : **in** Index_Type;
New_Item : **in** Element_Type)
with Pre => (**not** Tampering_With_Elements_Prohibited (Container)
or else raise Program_Error) **and then**
(Index **in**
First_Index (Container) .. Last_Index (Container)
or else raise Constraint_Error);

30/5 {*AI12-0112-1*} **procedure** Replace_Element (Container : **in out** Vector;
Position : **in** Cursor;
New_item : **in** Element_Type)
with Pre => (**not** Tampering_With_Elements_Prohibited (Container)
or else raise Program_Error) **and then**
(Position /= No_Element
or else raise Constraint_Error) **and then**
(Has_Element (Container, Position)
or else raise Program_Error);

31/5 {*AI12-0112-1*} **procedure** Query_Element
(Container : **in** Vector;
Index : **in** Index_Type;
Process : **not null access procedure** (Element : **in** Element_Type))
with Pre => Index **in**
First_Index (Container) .. Last_Index (Container)
or else raise Constraint_Error;

32/5 {*AI12-0112-1*} **procedure** Query_Element
(Position : **in** Cursor;
Process : **not null access procedure** (Element : **in** Element_Type))
with Pre => Position /= No_Element **or else raise** Constraint_Error,
Global => **in all**;

32.1/5 {*AI12-0112-1*} **procedure** Query_Element
(Container : **in** Vector;
Position : **in** Cursor;
Process : **not null access procedure** (Element : **in** Element_Type))
with Pre => (Position /= No_Element
or else raise Constraint_Error) **and then**
(Has_Element (Container, Position)
or else raise Program_Error);

33/5 {*AI12-0112-1*} **procedure** Update_Element
(Container : **in out** Vector;
Index : **in** Index_Type;
Process : **not null access procedure**
(Element : **in out** Element_Type))
with Pre => Index **in**
First_Index (Container) .. Last_Index (Container)
or else raise Constraint_Error;

34/5 {*AI12-0112-1*} **procedure** Update_Element
(Container : **in out** Vector;
Position : **in** Cursor;
Process : **not null access procedure**
(Element : **in out** Element_Type))
with Pre => (Position /= No_Element
or else raise Constraint_Error) **and then**
(Has_Element (Container, Position)
or else raise Program_Error);

34.1/5 {*AI05-0212-1*} {*AI12-0112-1*} **type** Constant_Reference_Type
(Element : **not null access constant** Element_Type) **is private**
with Implicit_Dereference => Element,
Nonblocking, Global => **in out synchronized**,
Default_Initial_Condition => (**raise** Program_Error);

34.a/5 **Discussion:** {*AI12-0112-1*} Finalization of this type will update the tampering counter of an associated container. We know this has to be an object of type Vector, but we don't have a way to specify that. We need this separate Global in case an object of this type is declared to exist separately from the short-lived object associated with a call of the Constant_Reference function.

```
{AI05-0212-1} {AI12-0112-1}    type Reference_Type (Element : not null access Element_Type) is private                    34.2/5
      with Implicit_Dereference => Element,
           Nonblocking, Global => in out synchronized,
           Default_Initial_Condition => (raise Program_Error);

{AI05-0212-1} {AI12-0112-1}    function Constant_Reference (Container : aliased in Vector;                               34.3/5
                                Index     : in Index_Type)
      return Constant_Reference_Type
      with Pre    => Index in
                        First_Index (Container) .. Last_Index (Container)
                        or else raise Constraint_Error,
           Post   => Tampering_With_Cursors_Prohibited (Container),
           Nonblocking, Global => null, Use_Formal => null;

{AI05-0212-1} {AI12-0112-1}    function Reference (Container : aliased in out Vector;                                    34.4/5
                       Index     : in Index_Type)
      return Reference_Type
      with Pre    => Index in
                        First_Index (Container) .. Last_Index (Container)
                        or else raise Constraint_Error,
           Post   => Tampering_With_Cursors_Prohibited (Container),
           Nonblocking, Global => null, Use_Formal => null;

{AI05-0212-1} {AI12-0112-1}    function Constant_Reference (Container : aliased in Vector;                               34.5/5
                                Position  : in Cursor)
      return Constant_Reference_Type
      with Pre  => (Position /= No_Element
                        or else raise Constraint_Error) and then
                    (Has_Element (Container, Position)
                        or else raise Program_Error),
           Post   => Tampering_With_Cursors_Prohibited (Container),
           Nonblocking, Global => null, Use_Formal => null;

{AI05-0212-1} {AI12-0112-1}    function Reference (Container : aliased in out Vector;                                    34.6/5
                       Position  : in Cursor)
      return Reference_Type
      with Pre  => (Position /= No_Element
                        or else raise Constraint_Error) and then
                    (Has_Element (Container, Position)
                        or else raise Program_Error),
           Post   => Tampering_With_Cursors_Prohibited (Container),
           Nonblocking, Global => null, Use_Formal => null;

{AI05-0001-1} {AI12-0112-1}    procedure Assign (Target : in out Vector; Source : in Vector)                             34.7/5
      with Pre  => not Tampering_With_Cursors_Prohibited (Target)
                      or else raise Program_Error,
           Post => Length (Source) = Length (Target) and then
                   Capacity (Target) >= Length (Target);

{AI05-0001-1} {AI12-0112-1}    function Copy (Source : Vector; Capacity : Count_Type := 0)                               34.8/5
      return Vector
      with Pre => Capacity = 0 or else Capacity >= Length (Source)
                      or else raise Capacity_Error,
           Post => Length (Copy'Result) = Length (Source) and then
                   not Tampering_With_Elements_Prohibited (Copy'Result)
                      and then
                   not Tampering_With_Cursors_Prohibited (Copy'Result)
                      and then
                   Copy'Result.Capacity >= (if Capacity = 0 then
                      Length (Source) else Capacity);
```

35/5 {*AI12-0112-1*}
```
   procedure Move (Target : in out Vector;
                   Source : in out Vector)
      with Pre  => (not Tampering_With_Cursors_Prohibited (Target)
                       or else raise Program_Error) and then
                   (not Tampering_With_Cursors_Prohibited (Source)
                       or else raise Program_Error),
           Post => (if not Target'Has_Same_Storage (Source) then
                       Length (Target) = Length (Source)'Old and then
                       Length (Source) = 0 and then
                       Capacity (Target) >= Length (Source)'Old);
```

36/5 {*AI12-0112-1*} {*AI12-0400-1*}
```
   procedure Insert_Vector (Container : in out Vector;
                            Before    : in     Extended_Index;
                            New_Item  : in     Vector)
      with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                      or else raise Program_Error) and then
                   (Before in
                      First_Index (Container) .. Last_Index (Container) + 1
                      or else raise Constraint_Error) and then
                   (Length (Container) <= Maximum_Length - Length (New_Item)
                      or else raise Constraint_Error),
           Post => Length (Container)'Old + Length (New_Item) =
                      Length (Container) and then
                   Capacity (Container) >= Length (Container);
```

37/5 {*AI12-0112-1*} {*AI12-0400-1*}
```
   procedure Insert_Vector (Container : in out Vector;
                            Before    : in     Cursor;
                            New_Item  : in     Vector)
      with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                      or else raise Program_Error) and then
                   (Before = No_Element or else
                    Has_Element (Container, Before)
                      or else raise Program_Error) and then
                   (Length (Container) <= Maximum_Length - Length (New_Item)
                      or else raise Constraint_Error),
           Post => Length (Container)'Old + Length (New_Item) =
                      Length (Container) and then
                   Capacity (Container) >= Length (Container);
```

38/5 {*AI12-0112-1*} {*AI12-0400-1*}
```
   procedure Insert_Vector (Container : in out Vector;
                            Before    : in     Cursor;
                            New_Item  : in     Vector;
                            Position  :    out Cursor)
      with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                      or else raise Program_Error) and then
                   (Before = No_Element or else
                    Has_Element (Container, Before)
                      or else raise Program_Error) and then
                   (Length (Container) <= Maximum_Length - Length (New_Item)
                      or else raise Constraint_Error),
           Post => Length (Container)'Old + Length (New_Item) =
                      Length (Container) and then
                   Has_Element (Container, Position) and then
                   Capacity (Container) >= Length (Container);
```

39/5 {*AI12-0112-1*}
```
   procedure Insert (Container : in out Vector;
                     Before    : in     Extended_Index;
                     New_Item  : in     Element_Type;
                     Count     : in     Count_Type := 1)
      with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                      or else raise Program_Error) and then
                   (Before in
                      First_Index (Container) .. Last_Index (Container) + 1
                      or else raise Constraint_Error) and then
                   (Length (Container) <= Maximum_Length - Count
                      or else raise Constraint_Error),
           Post => Length (Container)'Old + Count =
                      Length (Container) and then
                   Capacity (Container) >= Length (Container);
```

```
{AI12-0112-1}    procedure Insert (Container : in out Vector;                    40/5
                     Before    : in     Cursor;
                     New_Item  : in     Element_Type;
                     Count     : in     Count_Type := 1)
      with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                     or else raise Program_Error) and then
                   (Before = No_Element or else
                    Has_Element (Container, Before)
                     or else raise Program_Error) and then
                   (Length (Container) <= Maximum_Length - Count
                     or else raise Constraint_Error),
           Post => Length (Container)'Old + Count =
                     Length (Container) and then
                   Capacity (Container) >= Length (Container);

{AI12-0112-1}    procedure Insert (Container : in out Vector;                    41/5
                     Before    : in     Cursor;
                     New_Item  : in     Element_Type;
                     Position  :    out Cursor;
                     Count     : in     Count_Type := 1)
      with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                     or else raise Program_Error) and then
                   (Before = No_Element or else
                    Has_Element (Container, Before)
                     or else raise Program_Error) and then
                   (Length (Container) <= Maximum_Length - Count
                     or else raise Constraint_Error),
           Post => Length (Container)'Old + Count =
                     Length (Container) and then
                   Has_Element (Container, Position) and then
                   Capacity (Container) >= Length (Container);

{AI12-0112-1}    procedure Insert (Container : in out Vector;                    42/5
                     Before    : in     Extended_Index;
                     Count     : in     Count_Type := 1)
      with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                     or else raise Program_Error) and then
                   (Before in
                     First_Index (Container) .. Last_Index (Container) + 1
                     or else raise Constraint_Error) and then
                   (Length (Container) <= Maximum_Length - Count
                     or else raise Constraint_Error),
           Post => Length (Container)'Old + Count =
                     Length (Container) and then
                   Capacity (Container) >= Length (Container);

{AI12-0112-1}    procedure Insert (Container : in out Vector;                    43/5
                     Before    : in     Cursor;
                     Position  :    out Cursor;
                     Count     : in     Count_Type := 1)
      with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                     or else raise Program_Error) and then
                   (Before = No_Element or else
                    Has_Element (Container, Before)
                     or else raise Program_Error) and then
                   (Length (Container) <= Maximum_Length - Count
                     or else raise Constraint_Error),
           Post => Length (Container)'Old + Count = Length (Container)
                   and then Has_Element (Container, Position) and then
                   Capacity (Container) >= Length (Container);

{AI12-0112-1} {AI12-0400-1}    procedure Prepend_Vector (Container : in out Vector;    44/5
                               New_Item  : in     Vector)
      with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                     or else raise Program_Error) and then
                   (Length (Container) <= Maximum_Length - Length (New_Item)
                     or else raise Constraint_Error),
           Post => Length (Container)'Old + Length (New_Item) =
                     Length (Container) and then
                   Capacity (Container) >= Length (Container);
```

45/5 {*AI12-0112-1*}
```
procedure Prepend (Container : in out Vector;
                   New_Item  : in     Element_Type;
                   Count     : in     Count_Type := 1)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                   or else raise Program_Error) and then
                 (Length (Container) <= Maximum_Length - Count
                   or else raise Constraint_Error),
        Post => Length (Container)'Old + Count =
                   Length (Container) and then
                 Capacity (Container) >= Length (Container);
```

46/5 {*AI12-0112-1*} {*AI12-0400-1*}
```
procedure Append_Vector (Container : in out Vector;
                         New_Item  : in     Vector)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                   or else raise Program_Error) and then
                 (Length (Container) <= Maximum_Length - Length (New_Item)
                   or else raise Constraint_Error),
        Post => Length (Container)'Old + Length (New_Item) =
                   Length (Container) and then
                 Capacity (Container) >= Length (Container);
```

47/5 {*AI12-0112-1*} {*AI12-0400-1*}
```
procedure Append (Container : in out Vector;
                  New_Item  : in     Element_Type;
                  Count     : in     Count_Type)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error) and then
                 (Length (Container) <= Maximum_Length - Count
                    or else raise Constraint_Error),
        Post =>
           Length (Container)'Old + Count = Length (Container) and then
           Capacity (Container) >= Length (Container);
```

47.1/5 {*AI12-0112-1*} {*AI12-0212-1*} {*AI12-0400-1*}
```
procedure Append (Container : in out Vector;
                  New_Item  : in     Element_Type)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error) and then
                 (Length (Container) <= Maximum_Length - 1
                    or else raise Constraint_Error),
        Post => Length (Container)'Old + 1 = Length (Container) and then
                Capacity (Container) >= Length (Container);
```

48/5 {*AI12-0112-1*}
```
procedure Insert_Space (Container : in out Vector;
                        Before    : in     Extended_Index;
                        Count     : in     Count_Type := 1)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                   or else raise Program_Error) and then
                 (Before in
                    First_Index (Container) .. Last_Index (Container) + 1
                    or else raise Constraint_Error) and then
                 (Length (Container) <= Maximum_Length - Count
                    or else raise Constraint_Error),
        Post => Length (Container)'Old + Count =
                   Length (Container) and then
                 Capacity (Container) >= Length (Container);
```

```
{AI12-0112-1}    procedure Insert_Space (Container : in out Vector;                    49/5
                          Before    : in     Cursor;
                          Position  :    out Cursor;
                          Count     : in     Count_Type := 1)
      with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                       or else raise Program_Error) and then
                    (Before = No_Element or else
                       Has_Element (Container, Before)
                       or else raise Program_Error) and then
                    (Length (Container) <= Maximum_Length - Count
                       or else raise Constraint_Error),
           Post => Length (Container)'Old + Count =
                       Length (Container) and then
                    Has_Element (Container, Position) and then
                    Capacity (Container) >= Length (Container);

{AI12-0112-1}    procedure Delete (Container : in out Vector;                          50/5
                       Index     : in     Extended_Index;
                       Count     : in     Count_Type := 1)
      with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                       or else raise Program_Error) and then
                    (Index in
                       First_Index (Container) .. Last_Index (Container) + 1
                       or else raise Constraint_Error),
           Post => Length (Container)'Old - Count <=
                       Length (Container);

{AI12-0112-1}    procedure Delete (Container : in out Vector;                          51/5
                       Position  : in out Cursor;
                       Count     : in     Count_Type := 1)
      with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                       or else raise Program_Error) and then
                    (Position /= No_Element
                       or else raise Constraint_Error) and then
                    (Has_Element (Container, Position)
                       or else raise Program_Error),
           Post => Length (Container)'Old - Count <=
                       Length (Container) and then
                    Position = No_Element;

{AI12-0112-1}    procedure Delete_First (Container : in out Vector;                    52/5
                          Count     : in     Count_Type := 1)
      with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                       or else raise Program_Error,
           Post => Length (Container)'Old - Count <= Length (Container);

{AI12-0112-1}    procedure Delete_Last (Container : in out Vector;                     53/5
                       Count     : in     Count_Type := 1)
      with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                       or else raise Program_Error,
           Post => Length (Container)'Old - Count <= Length (Container);

{AI12-0112-1}    procedure Reverse_Elements (Container : in out Vector)               54/5
      with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                       or else raise Program_Error;

{AI12-0112-1}    procedure Swap (Container : in out Vector;                           55/5
                    I, J      : in     Index_Type)
      with Pre  => (not Tampering_With_Elements_Prohibited (Container)
                       or else raise Program_Error) and then
                   (I in First_Index (Container) .. Last_Index (Container)
                       or else raise Constraint_Error) and then
                   (J in First_Index (Container) .. Last_Index (Container)
                       or else raise Constraint_Error);
```

```
56/5    {AI12-0112-1}    procedure Swap (Container : in out Vector;
                           I, J      : in     Cursor)
              with Pre  => (not Tampering_With_Elements_Prohibited (Container)
                               or else raise Program_Error) and then
                           (I /= No_Element or else Constraint_Error) and then
                           (J /= No_Element or else Constraint_Error) and then
                           (Has_Element (Container, I)
                               or else raise Program_Error) and then
                           (Has_Element (Container, J)
                               or else raise Program_Error);

57/5    {AI12-0112-1}    function First_Index (Container : Vector) return Index_Type
              with Nonblocking, Global => null, Use_Formal => null,
                   Post => First_Index'Result = Index_Type'First;

58/5    {AI12-0112-1}    function First (Container : Vector) return Cursor
              with Nonblocking, Global => null, Use_Formal => null,
                   Post => (if not Is_Empty (Container)
                            then Has_Element (Container, First'Result)
                            else First'Result = No_Element);

59/5    {AI12-0112-1}    function First_Element (Container : Vector)
              return Element_Type
              with Pre => (not Is_Empty (Container)
                              or else raise Constraint_Error);

60/5    {AI12-0112-1}    function Last_Index (Container : Vector) return Extended_Index
              with Nonblocking, Global => null, Use_Formal => null,
                   Post => (if Length (Container) = 0
                            then Last_Index'Result = No_Index
                            else Count_Type(Last_Index'Result - Index_Type'First) =
                                 Length (Container) - 1);

61/5    {AI12-0112-1}    function Last (Container : Vector) return Cursor
              with Nonblocking, Global => null, Use_Formal => null,
                   Post => (if not Is_Empty (Container)
                            then Has_Element (Container, Last'Result)
                            else Last'Result = No_Element);

62/5    {AI12-0112-1}    function Last_Element (Container : Vector)
              return Element_Type
              with Pre => (not Is_Empty (Container)
                              or else raise Constraint_Error);

63/5    {AI12-0112-1}    function Next (Position : Cursor) return Cursor
              with Nonblocking, Global => in all, Use_Formal => null,
                   Post => (if Position = No_Element then Next'Result = No_Element);

63.1/5  {AI12-0112-1}    function Next (Container : Vector; Position : Cursor) return
        Cursor
              with Nonblocking, Global => null, Use_Formal => null,
                   Pre  => Position = No_Element or else
                           Has_Element (Container, Position)
                               or else raise Program_Error,
                   Post => (if Position = No_Element then Next'Result = No_Element
                            elsif Has_Element (Container, Next'Result) then
                               To_Index (Container, Next'Result) =
                               To_Index (Container, Position) + 1
                            elsif Next'Result = No_Element then
                               Position = Last (Container)
                            else False);

64/5    {AI12-0112-1}    procedure Next (Position : in out Cursor)
              with Nonblocking, Global => in all, Use_Formal => null;

64.1/5  {AI12-0112-1}    procedure Next (Container : in     Vector;
                           Position  : in out Cursor)
              with Nonblocking, Global => null, Use_Formal => null,
                   Pre  => Position = No_Element or else
                           Has_Element (Container, Position)
                               or else raise Program_Error,
                   Post => (if Position /= No_Element
                            then Has_Element (Container, Position));
```

```
{AI12-0112-1}    function Previous (Position : Cursor) return Cursor                    65/5
      with Nonblocking, Global => in all, Use_Formal => null,
           Post => (if Position = No_Element
                    then Previous'Result = No_Element);

{AI12-0112-1}    function Previous (Container : Vector;                                 65.1/5
                                    Position  : Cursor) return Cursor
      with Nonblocking, Global => null, Use_Formal => null,
           Pre  => Position = No_Element or else
                   Has_Element (Container, Position)
                       or else raise Program_Error,
           Post => (if Position = No_Element
                    then Previous'Result = No_Element
                    elsif Has_Element (Container, Previous'Result) then
                       To_Index (Container, Previous'Result) =
                       To_Index (Container, Position) - 1
                    elsif Previous'Result = No_Element then
                       Position = First (Container)
                    else False);

{AI12-0112-1}    procedure Previous (Position : in out Cursor)                          66/5
      with Nonblocking, Global => in all, Use_Formal => null;

{AI12-0112-1}    procedure Previous (Container : in     Vector;                         66.1/5
                                    Position  : in out Cursor)
      with Nonblocking, Global => null, Use_Formal => null,
           Pre  => Position = No_Element or else
                   Has_Element (Container, Position)
                       or else raise Program_Error,
           Post => (if Position /= No_Element
                    then Has_Element (Container, Position));

   function Find_Index (Container : Vector;                                           67/2
                        Item      : Element_Type;
                        Index     : Index_Type := Index_Type'First)
      return Extended_Index;

{AI12-0112-1}    function Find (Container : Vector;                                     68/5
                  Item      : Element_Type;
                  Position  : Cursor := No_Element)
      return Cursor
      with Pre  => Position = No_Element or else
                   Has_Element (Container, Position)
                       or else raise Program_Error,
           Post => (if Find'Result /= No_Element
                    then Has_Element (Container, Find'Result));

   function Reverse_Find_Index (Container : Vector;                                   69/2
                                Item      : Element_Type;
                                Index     : Index_Type := Index_Type'Last)
      return Extended_Index;

{AI12-0112-1}    function Reverse_Find (Container : Vector;                             70/5
                          Item      : Element_Type;
                          Position  : Cursor := No_Element)
      return Cursor
      with Pre  => Position = No_Element or else
                   Has_Element (Container, Position)
                       or else raise Program_Error,
           Post => (if Reverse_Find'Result /= No_Element
                    then Has_Element (Container, Reverse_Find'Result));

   function Contains (Container : Vector;                                             71/2
                      Item      : Element_Type) return Boolean;

This paragraph was deleted.{AI05-0212-1}                                              72/3

{AI12-0112-1}    procedure Iterate                                                     73/5
     (Container : in Vector;
      Process   : not null access procedure (Position : in Cursor))
      with Allows_Exit;
```

74/5 {*AI12-0112-1*} **procedure** Reverse_Iterate
(Container : **in** Vector;
Process : **not null access procedure** (Position : **in** Cursor))
with Allows_Exit;

74.1/5 {*AI05-0212-1*} {*AI12-0112-1*} {*AI12-0266-1*} **function** Iterate (Container : **in** Vector)
return Vector_Iterator_Interfaces.Parallel_Reversible_Iterator'Class
with Post => Tampering_With_Cursors_Prohibited (Container);

74.2/5 {*AI05-0212-1*} {*AI12-0112-1*} **function** Iterate (Container : **in** Vector; Start : **in** Cursor)
return Vector_Iterator_Interfaces.Reversible_Iterator'Class
with Pre => (Start /= No_Element
or else raise Constraint_Error) **and then**
(Has_Element (Container, Start)
or else raise Program_Error),
Post => Tampering_With_Cursors_Prohibited (Container);

75/5 {*AI12-0112-1*} **generic**
with function "<" (Left, Right : Element_Type)
return Boolean is <>;
package Generic_Sorting
with Nonblocking, Global => **null is**

76/2 **function** Is_Sorted (Container : Vector) **return** Boolean;

77/5 {*AI12-0112-1*} **procedure** Sort (Container : **in out** Vector)
with Pre => **not** Tampering_With_Cursors_Prohibited (Container)
or else raise Program_Error;

78/5 {*AI12-0112-1*} **procedure** Merge (Target : **in out** Vector;
Source : **in out** Vector)
with Pre => (**not** Tampering_With_Cursors_Prohibited (Target)
or else raise Program_Error) **and then**
(**not** Tampering_With_Cursors_Prohibited (Source)
or else raise Program_Error) **and then**
(Length (Target) <= Maximum_Length - Length (Source)
or else raise Constraint_Error) **and then**
((Length (Source) = 0 **or else**
not Target'Has_Same_Storage (Source))
or else raise Program_Error),
Post => (**declare**
Result_Length : **constant** Count_Type :=
Length (Source)'Old + Length (Target)'Old;
begin
(Length (Source) = 0 **and then**
Length (Target) = Result_Length **and then**
Capacity (Target) >= Result_Length));

79/2 **end** Generic_Sorting;

79.1/5 {*AI12-0111-1*} **package** Stable **is**

79.2/5 {*AI12-0111-1*} {*AI12-0339-1*} {*AI12-0400-1*} {*AI12-0407-1*} **type** Vector (Base : **not null access** Vectors.Vector) **is**
tagged limited private
with Constant_Indexing => Constant_Reference,
Variable_Indexing => Reference,
Default_Iterator => Iterate,
Iterator_Element => Element_Type,
Stable_Properties => (Length, Capacity),
Global => **null**,
Default_Initial_Condition => Length (Vector) = 0,
Preelaborable_Initialization;

79.a/5 **Discussion:** {*AI05-0112-1*} The Global of **null** assumes that the user of a stable object is including effects associated with the access discriminant. For operations with **in** parameters (after any overriding), the object designated by the access discriminant is assumed to be read, and for other operations (including initialization and finalization) the object designated by the access discriminant is assumed to be read and updated.

79.3/5 {*AI12-0111-1*} **type** Cursor **is private**
with Preelaborable_Initialization;

```
{AI12-0111-1}       Empty_Vector : constant Vector;                                    79.4/5
{AI12-0111-1}       No_Element : constant Cursor;                                      79.5/5
{AI12-0111-1}       function Has_Element (Position : Cursor) return Boolean            79.6/5
        with Nonblocking, Global => in all, Use_Formal => null;
{AI12-0111-1}       package Vector_Iterator_Interfaces is new                          79.7/5
        Ada.Iterator_Interfaces (Cursor, Has_Element);
{AI12-0111-1}       procedure Assign (Target : in out Vectors.Vector;                  79.8/5
                                      Source : in Vector)
        with Post => Length (Source) = Length (Target) and then
                     Capacity (Target) >= Length (Target);
{AI12-0111-1}       function Copy (Source : Vectors.Vector) return Vector              79.9/5
        with Post => Length (Copy'Result) = Length (Source);
{AI12-0111-1}       type Constant_Reference_Type                                       79.10/5
           (Element : not null access constant Element_Type) is private
        with Implicit_Dereference => Element,
             Nonblocking, Global => null, Use_Formal => null,
             Default_Initial_Condition => (raise Program_Error);
{AI12-0111-1}       type Reference_Type                                                79.11/5
           (Element : not null access Element_Type) is private
        with Implicit_Dereference => Element,
             Nonblocking, Global => null, Use_Formal => null,
             Default_Initial_Condition => (raise Program_Error);
{AI12-0111-1}       -- Additional subprograms as described in the text                 79.12/5
      -- are declared here.
{AI12-0111-1}    private                                                               79.13/5
{AI12-0111-1}       ... -- not specified by the language                               79.14/5
{AI12-0111-1}    end Stable;                                                           79.15/5
private                                                                               80/2
   ... -- not specified by the language                                               81/2
end Ada.Containers.Vectors;                                                           82/2
```

{*AI95-00302-03*} The actual function for the generic formal function "=" on Element_Type values is expected to define a reflexive and symmetric relationship and return the same result value each time it is called with a particular pair of values. If it behaves in some other manner, the functions defined to use it return an unspecified value. The exact arguments and number of calls of this generic formal function by the functions defined to use it are unspecified. 83/2

Ramification: The “functions defined to use it” are Find, Find_Index, Reverse_Find, Reverse_Find_Index, and "=" for Vectors. This list is a bit too long to give explicitly. 83.a/2

If the actual function for "=" is not symmetric and consistent, the result returned by any of the functions defined to use "=" cannot be predicted. The implementation is not required to protect against "=" raising an exception, or returning random results, or any other “bad” behavior. And it can call "=" in whatever manner makes sense. But note that only the results of the functions defined to use "=" are unspecified; other subprograms are not allowed to break if "=" is bad. 83.b/2

{*AI95-00302-03*} The type Vector is used to represent vectors. The type Vector needs finalization (see 7.6). 84/2

{*AI95-00302-03*} Empty_Vector represents the empty vector object. It has a length of 0. If an object of type Vector is not otherwise initialized, it is initialized to the same value as Empty_Vector. 85/2

{*AI95-00302-03*} No_Element represents a cursor that designates no element. If an object of type Cursor is not otherwise initialized, it is initialized to the same value as No_Element. 86/2

{*AI95-00302-03*} {*AI12-0434-1*} The primitive "=" operator for type Cursor returns True if both cursors are No_Element, or designate the same element in the same container. 87/5

To be honest: {*AI12-0434-1*} “The primitive "=" operator” is the one with two parameters of type Cursor which returns Boolean. We're not talking about some other (hidden) primitive function named "=". 87.a/5

88/2 {*AI95-00302-03*} Execution of the default implementation of the Input, Output, Read, or Write attribute of type Cursor raises Program_Error.

88.a/2 **Reason:** A cursor will probably be implemented in terms of one or more access values, and the effects of streaming access values is unspecified. Rather than letting the user stream junk by accident, we mandate that streaming of cursors raise Program_Error by default. The attributes can always be specified if there is a need to support streaming.

88.1/5 {*AI05-0001-1*} {*AI05-0262-1*} {*AI12-0437-1*} Vector'Write for a Vector object *V* writes Length(*V*) elements of the vector to the stream. It may also write additional information about the vector.

88.2/3 {*AI05-0001-1*} {*AI05-0262-1*} Vector'Read reads the representation of a vector from the stream, and assigns to *Item* a vector with the same length and elements as was written by Vector'Write.

88.b/3 **Implementation Note:** The Reference Manual requires streaming of all language-defined nonlimited types (including containers) to "work" (see 13.13.2). In addition, we do not want all of the elements that make up the capacity of the vector streamed, as those beyond the length of the container have undefined contents (and might cause bad things when read back in). This will require a custom stream attribute implementation; the language-defined default implementation will not work (even for a bounded form, as that would most likely stream the entire capacity of the vector). There is a separate requirement that the unbounded and Bounded form use the same streaming representation for the same element type, see A.18.19.

89/2 {*AI95-00302-03*} No_Index represents a position that does not correspond to any element. The subtype Extended_Index includes the indices covered by Index_Type plus the value No_Index and, if it exists, the successor to the Index_Type'Last.

89.a/2 **Discussion:** We require the existence of Index_Type'First – 1, so that No_Index and Last_Index of an empty vector is well-defined. We don't require the existence of Index_Type'Last + 1, as it is only used as the position of insertions (and needs to be allowed only when inserting an empty vector).

90/5 {*AI95-00302-03*} {*AI12-0111-1*} {*AI12-0112-1*} [Some operations check for “tampering with cursors” of a container because they depend on the set of elements of the container remaining constant, and others check for “tampering with elements” of a container because they depend on elements of the container not being replaced.] When tampering with cursors is *prohibited* for a particular vector object *V*, Program_Error is propagated by the finalization of *V*[, as well as by a call that passes *V* to certain of the operations of this package, as indicated by the precondition of such an operation]. Similarly, when tampering with elements is *prohibited* for *V*, Program_Error is propagated by a call that passes *V* to certain of the other operations of this package, as indicated by the precondition of such an operation.

Paragraphs 91 through 97 are removed as preconditions now describe these rules.

93.a.1/3 **Ramification:** We don't need to explicitly mention assignment_statement, because that finalizes the target object as part of the operation, and finalization of an object is already defined as tampering with cursors.

97.2/5
```
function Has_Element (Position : Cursor) return Boolean
   with Nonblocking, Global => in all, Use_Formal => null;
```

97.3/3 {*AI05-0212-1*} Returns True if Position designates an element, and returns False otherwise.

97.c/3 **To be honest:** {*AI05-0005-1*} {*AI05-0212-1*} This function might not detect cursors that designate deleted elements; such cursors are invalid (see below) and the result of calling Has_Element with an invalid cursor is unspecified (but not erroneous).

97.4/5
```
function Has_Element (Container : Vector; Position : Cursor)
   return Boolean
   with Nonblocking, Global => null, Use_Formal => null;
```

97.5/5 {*AI12-0112-1*} Returns True if Position designates an element in Container, and returns False otherwise.

97.d/5 **Ramification:** If Position is No_Element, Has_Element returns False.

98/2
```
function "=" (Left, Right : Vector) return Boolean;
```

99/3 {*AI95-00302-03*} {*AI05-0264-1*} If Left and Right denote the same vector object, then the function returns True. If Left and Right have different lengths, then the function returns False.

Otherwise, it compares each element in Left to the corresponding element in Right using the generic formal equality operator. If any such comparison returns False, the function returns False; otherwise, it returns True. Any exception raised during evaluation of element equality is propagated.

Implementation Note: This wording describes the canonical semantics. However, the order and number of calls on the formal equality function is unspecified for all of the operations that use it in this package, so an implementation can call it as many or as few times as it needs to get the correct answer. Specifically, there is no requirement to call the formal equality additional times once the answer has been determined. 99.a/2

```
function Tampering_With_Cursors_Prohibited
   (Container : Vector) return Boolean
   with Nonblocking, Global => null, Use_Formal => null;
```
99.1/5

{*AI12-0112-1*} Returns True if tampering with cursors or tampering with elements is currently prohibited for Container, and returns False otherwise. 99.2/5

Reason: {*AI12-0112-1*} Prohibiting tampering with elements also needs to prohibit tampering with cursors, as deleting an element is similar to replacing it. 99.b/5

Implementation Note: {*AI12-0112-1*} Various contracts elsewhere in this specification require that this function be implemented with synchronized data. Moreover, it is possible for tampering to be prohibited by multiple operations (sequentially or in parallel). Therefore, tampering needs to be implemented with an atomic or protected counter. The counter is initialized to zero, and is incremented when tampering is prohibited, and decremented when leaving an area that prohibited tampering. Function Tampering_With_Cursors_Prohibited returns True if the counter is nonzero. (Note that any case where the result is not well-defined for one task is incorrect use of shared variables and would be erroneous by the rules of 9.10, so no special protection is needed to read the counter.) 99.c/5

```
function Tampering_With_Elements_Prohibited
   (Container : Vector) return Boolean
   with Nonblocking, Global => null, Use_Formal => null;
```
99.3/5

{*AI12-0112-1*} Always returns False[, regardless of whether tampering with elements is prohibited]. 99.4/5

Reason: {*AI12-0111-1*} A definite element cannot change size, so we allow operations that tamper with elements even when tampering with elements is prohibited. That's not true for the indefinite containers, which is why this kind of tampering exists. 99.d/5

```
function Maximum_Length return Count_Type
   with Nonblocking, Global => null, Use_Formal => null;
```
99.5/5

{*AI12-0112-1*} Returns the maximum Length of a Vector, based on the index type. 99.6/5

Implementation Note: This is just: 99.e/5

```
Count_Type (Index_Type'Last - Index_Type'First + 1)
```
99.f

but since the inner calculation can overflow or the type conversion can fail, this can't be evaluated in general with an expression function. Note that if this expression raises Constraint_Error, then the result is Count_Type'Last, since the Capacity of a Vector cannot exceed Count_Type'Last. 99.g/5

```
function Empty (Capacity : Count_Type := implementation-defined)
   return Vector
   with Pre  => Capacity <= Maximum_Length
                   or else raise Constraint_Error,
        Post =>
           Capacity (Empty'Result) >= Capacity and then
           not Tampering_With_Elements_Prohibited (Empty'Result) and then
           not Tampering_With_Cursors_Prohibited (Empty'Result) and then
           Length (Empty'Result) = 0;
```
99.7/5

{*AI12-0339-1*} Returns an empty vector. 99.8/5

100/5
```
function To_Vector (Length : Count_Type) return Vector
   with Pre  => Length <= Maximum_Length or else raise Constraint_Error,
        Post =>
           To_Vector'Result.Length = Length and then
           not Tampering_With_Elements_Prohibited (To_Vector'Result)
             and then
           not Tampering_With_Cursors_Prohibited (To_Vector'Result)
             and then
           To_Vector'Result.Capacity >= Length;
```

101/2 {*AI95-00302-03*} Returns a vector with a length of Length, filled with empty elements.

102/5
```
function To_Vector
  (New_Item : Element_Type;
   Length   : Count_Type) return Vector
   with Pre  => Length <= Maximum_Length or else raise Constraint_Error,
        Post =>
           To_Vector'Result.Length = Length and then
           not Tampering_With_Elements_Prohibited (To_Vector'Result)
             and then
           not Tampering_With_Cursors_Prohibited (To_Vector'Result)
             and then
        To_Vector'Result.Capacity >= Length;
```

103/2 {*AI95-00302-03*} Returns a vector with a length of Length, filled with elements initialized to the value New_Item.

104/5
```
function "&" (Left, Right : Vector) return Vector
   with Pre  => Length (Left) <= Maximum_Length - Length (Right)
                  or else raise Constraint_Error,
        Post => Length (Vectors."&"'Result) =
                   Length (Left) + Length (Right) and then
                not Tampering_With_Elements_Prohibited (Vectors."&"'Result)
                   and then
                not Tampering_With_Cursors_Prohibited (Vectors."&"'Result)
                   and then
                Vectors."&"'Result.Capacity >=
                   Length (Left) + Length (Right);
```

105/2 {*AI95-00302-03*} Returns a vector comprising the elements of Left followed by the elements of Right.

106/5
```
function "&" (Left  : Vector;
              Right : Element_Type) return Vector
   with Pre  => Length (Left) <= Maximum_Length - 1
                   or else raise Constraint_Error,
        Post => Vectors."&"'Result.Length = Length (Left) + 1 and then
                not Tampering_With_Elements_Prohibited (Vectors."&"'Result)
                   and then
                not Tampering_With_Cursors_Prohibited (Vectors."&"'Result)
                   and then
                Vectors."&"'Result.Capacity >= Length (Left) + 1;
```

107/2 {*AI95-00302-03*} Returns a vector comprising the elements of Left followed by the element Right.

```
function "&" (Left  : Element_Type;                                                    108/5
              Right : Vector) return Vector
   with Pre  => Length (Right) <= Maximum_Length - 1
                   or else raise Constraint_Error,
        Post => Length (Vectors."&"'Result) = Length (Right) + 1 and then
                not Tampering_With_Elements_Prohibited (Vectors."&"'Result)
                   and then
                not Tampering_With_Cursors_Prohibited (Vectors."&"'Result)
                   and then
                Vectors."&"'Result.Capacity >= Length (Right) + 1;
```

{*AI95-00302-03*} Returns a vector comprising the element Left followed by the elements of Right. 109/2

```
function "&" (Left, Right  : Element_Type) return Vector                               110/5
   with Pre  => Maximum_Length >= 2 or else raise Constraint_Error,
        Post => Length ("&"'Result) = 2 and then
                not Tampering_With_Elements_Prohibited (Vectors."&"'Result)
                   and then
                not Tampering_With_Cursors_Prohibited (Vectors."&"'Result)
                   and then
                Vectors."&"'Result.Capacity >= 2;
```

{*AI95-00302-03*} Returns a vector comprising the element Left followed by the element Right. 111/2

```
{AI12-0112-1} function Capacity (Container : Vector) return Count_Type                 112/5
   with Nonblocking, Global => null, Use_Formal => null;
```

{*AI95-00302-03*} Returns the capacity of Container. 113/2

```
{AI12-0112-1} procedure Reserve_Capacity (Container : in out Vector;                   114/5
                          Capacity  : in     Count_Type)
   with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                   or else raise Program_Error,
        Post => Container.Capacity >= Capacity;
```

{*AI95-00302-03*} {*AI05-0001-1*} {*AI05-0264-1*} If the capacity of Container is already greater than or equal to Capacity, then Reserve_Capacity has no effect. Otherwise, Reserve_Capacity allocates additional storage as necessary to ensure that the length of the resulting vector can become at least the value Capacity without requiring an additional call to Reserve_Capacity, and is large enough to hold the current length of Container. Reserve_Capacity then, as necessary, moves elements into the new storage and deallocates any storage no longer needed. Any exception raised during allocation is propagated and Container is not modified. 115/3

Discussion: Expanding the internal array can be done by allocating a new, longer array, copying the elements, and deallocating the original array. This may raise Storage_Error, or cause an exception from a controlled subprogram. We require that a failed Reserve_Capacity does not lose any elements if an exception occurs, but we do not require a specific order of evaluations or copying. 115.a/2

This routine is used to preallocate the internal array to the specified capacity such that future Inserts do not require memory allocation overhead. Therefore, the implementation should allocate the needed memory to make that true at this point, even though the visible semantics could be preserved by waiting until the memory is needed. This doesn't apply to the indefinite element container, because elements will have to be allocated individually. 115.b/2

The implementation does not have to contract the internal array if the capacity is reduced, as any capacity greater than or equal to the specified capacity is allowed. 115.c/2

```
function Length (Container : Vector) return Count_Type                                 116/5
   with Nonblocking, Global => null, Use_Formal => null;
```

{*AI95-00302-03*} Returns the number of elements in Container. 117/2

118/5
```
procedure Set_Length (Container : in out Vector;
                      Length    : in     Count_Type)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error) and then
                (Length <= Maximum_Length or else raise Constraint_Error),
        Post => Container.Length = Length and then
                Capacity (Container) >= Length;
```

119/3 {*AI95-00302-03*} {*AI05-0264-1*} If Length is larger than the capacity of Container, Set_Length calls Reserve_Capacity (Container, Length), then sets the length of the Container to Length. If Length is greater than the original length of Container, empty elements are added to Container; otherwise, elements are removed from Container.

119.a/2 **Ramification:** No elements are moved by this operation; any new empty elements are added at the end. This follows from the rules that a cursor continues to designate the same element unless the routine is defined to make the cursor ambiguous or invalid; this operation does not do that.

120/5
```
function Is_Empty (Container : Vector) return Boolean
   with Nonblocking, Global => null, Use_Formal => null,
        Post => Is_Empty'Result = (Length (Container) = 0);
```

121/5 {*AI95-00302-03*} {*AI12-0112-1*} Returns True if Container is empty.

122/5
```
procedure Clear (Container : in out Vector)
   with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error,
        Post => Length (Container) = 0;
```

123/2 {*AI95-00302-03*} Removes all the elements from Container. The capacity of Container does not change.

124/5
```
function To_Cursor (Container : Vector;
                    Index     : Extended_Index) return Cursor
   with Post => (if Index in
                    First_Index (Container) .. Last_Index (Container)
                 then Has_Element (Container, To_Cursor'Result)
                 else To_Cursor'Result = No_Element),
        Nonblocking, Global => null, Use_Formal => null;
```

125/5 {*AI95-00302-03*} {*AI12-0112-1*} {*AI12-0196-1*} Returns a cursor designating the element at position Index in Container; returns No_Element if Index does not designate an element. For the purposes of determining whether the parameters overlap in a call to To_Cursor, the Container parameter is not considered to overlap with any object [(including itself)].

125.a/5 **Reason:** {*AI12-0196-1*} Without the preceding rule, concurrent calls to To_Cursor on the same container would interfere by the concurrent call rules in Annex A, since the container object of the concurrent calls would overlap with itself. We want these to not interfere, for example to allow the Vector elements to be split into separate “chunks” for parallel processing.

126/5
```
function To_Index (Position  : Cursor) return Extended_Index
   with Nonblocking, Global => in all, Use_Formal => null;
```

127/2 {*AI95-00302-03*} If Position is No_Element, No_Index is returned. Otherwise, the index (within its containing vector) of the element designated by Position is returned.

127.a/2 **Ramification:** This implies that the index is determinable from a bare cursor alone. The basic model is that a vector cursor is implemented as a record containing an access to the vector container and an index value. This does constrain implementations, but it also allows all of the cursor operations to be defined in terms of the corresponding index operation (which should be primary for a vector).

```
function To_Index (Container : Vector;                                          127.1/5
                   Position  : Cursor) return Extended_Index
   with Pre  => Position = No_Element or else
                Has_Element (Container, Position)  or else
                   raise Program_Error,
        Post => (if Position = No_Element then To_Index'Result = No_Index
                 else To_Index'Result in First_Index (Container) ..
                      Last_Index (Container)),
        Nonblocking, Global => null, Use_Formal => null;
```

{*AI12-0112-1*} Returns the index (within Container) of the element designated by Position; returns No_Index if Position does not designate an element. For the purposes of determining whether the parameters overlap in a call to To_Index, the Container parameter is not considered to overlap with any object [(including itself)]. 127.2/5

```
function Element (Container : Vector;                                           128/5
                  Index     : Index_Type)
   return Element_Type
   with Pre  => Index in First_Index (Container) .. Last_Index (Container)
                   or else raise Constraint_Error,
        Nonblocking, Global => null, Use_Formal => Element_Type;
```

{*AI95-00302-03*} {*AI12-0112-1*} Element returns the element at position Index. 129/5

```
function Element (Position  : Cursor) return Element_Type                       130/5
   with Pre  => Position /= No_Element or else raise Constraint_Error,
        Nonblocking, Global => in all, Use_Formal => Element_Type;
```

{*AI95-00302-03*} {*AI12-0112-1*} Element returns the element designated by Position. 131/5

```
function Element (Container : Vector;                                           131.1/5
                  Position  : Cursor) return Element_Type
   with Pre => (Position /= No_Element
                   or else raise Constraint_Error) and then
               (Has_Element (Container, Position)
                   or else raise Program_Error),
        Nonblocking, Global => null, Use_Formal => Element_Type;
```

{*AI12-0112-1*} Element returns the element designated by Position in Container. 131.2/5

```
procedure Replace_Element (Container : in out Vector;                           132/5
                           Index     : in     Index_Type;
                           New_Item  : in     Element_Type)
   with Pre => (not Tampering_With_Elements_Prohibited (Container)
                   or else raise Program_Error) and then
               (Index in First_Index (Container) .. Last_Index (Container)
                   or else raise Constraint_Error);
```

{*AI95-00302-03*} {*AI05-0264-1*} {*AI12-0112-1*} {*AI12-0196-1*} Replace_Element assigns the value New_Item to the element at position Index. Any exception raised during the assignment is propagated. The element at position Index is not an empty element after successful call to Replace_Element. For the purposes of determining whether the parameters overlap in a call to Replace_Element, the Container parameter is not considered to overlap with any object [(including itself)], and the Index parameter is considered to overlap with the element at position Index. 133/5

134/5
```
procedure Replace_Element (Container : in out Vector;
                           Position  : in     Cursor;
                           New_Item  : in     Element_Type)
   with Pre  => (not Tampering_With_Elements_Prohibited (Container)
                    or else raise Program_Error) and then
                 (Position /= No_Element
                    or else raise Constraint_Error) and then
                 (Has_Element (Container, Position)
                    or else raise Program_Error);
```

135/5 {*AI95-00302-03*} {*AI05-0264-1*} {*AI12-0112-1*} {*AI12-0196-1*} Replace_Element assigns New_Item to the element designated by Position. Any exception raised during the assignment is propagated. The element at Position is not an empty element after successful call to Replace_Element. For the purposes of determining whether the parameters overlap in a call to Replace_Element, the Container parameter is not considered to overlap with any object [(including itself)].

135.a/3 **Ramification:** {*AI05-0212-1*} Replace_Element, Update_Element, and Reference are the only ways that an element can change from empty to nonempty. Also see the note following Update_Element.

136/5
```
procedure Query_Element
  (Container : in Vector;
   Index     : in Index_Type;
   Process   : not null access procedure (Element : in Element_Type))
   with Pre  => Index in First_Index (Container) .. Last_Index (Container)
                   or else raise Constraint_Error;
```

137/5 {*AI95-00302-03*} {*AI05-0265-1*} {*AI12-0112-1*} Query_Element calls Process.**all** with the element at position Index as the argument. Tampering with the elements of Container is prohibited during the execution of the call on Process.**all**. Any exception raised by Process.**all** is propagated.

137.a/2 **Reason:** {*AI05-0005-1*} The "tamper with the elements" check is intended to prevent the Element parameter of Process from being replaced or deleted outside of Process. The check prevents data loss (if Element_Type is passed by copy) or erroneous execution (if Element_Type is an unconstrained type in an indefinite container).

138/5
```
procedure Query_Element
  (Position : in Cursor;
   Process  : not null access procedure (Element : in Element_Type))
   with Pre  => Position /= No_Element or else raise Constraint_Error
        Global => in all;
```

139/5 {*AI95-00302-03*} {*AI05-0021-1*} {*AI05-0265-1*} {*AI12-0112-1*} Query_Element calls Process.**all** with the element designated by Position as the argument. Tampering with the elements of the vector that contains the element designated by Position is prohibited during the execution of the call on Process.**all**. Any exception raised by Process.**all** is propagated.

139.1/5
```
procedure Query_Element
  (Container : in Vector;
   Position  : in Cursor;
   Process   : not null access procedure (Element : in Element_Type))
   with Pre  => (Position /= No_Element
                    or else raise Constraint_Error) and then
                (Has_Element (Container, Position)
                    or else raise Program_Error);
```

139.2/5 {*AI12-0112-1*} Query_Element calls Process.**all** with the element designated by Position as the argument. Tampering with the elements of Container is prohibited during the execution of the call on Process.**all**. Any exception raised by Process.**all** is propagated.

```
procedure Update_Element                                                    140/5
  (Container : in out Vector;
   Index     : in     Index_Type;
   Process   : not null access procedure
                     (Element : in out Element_Type))
   with Pre  => Index in First_Index (Container) .. Last_Index (Container)
                     or else raise Constraint_Error;
```

{*AI95-00302-03*} {*AI05-0265-1*} {*AI12-0112-1*} Update_Element calls Process.**all** with the element at position Index as the argument. Tampering with the elements of Container is prohibited during the execution of the call on Process.**all**. Any exception raised by Process.**all** is propagated. 141/5

If Element_Type is unconstrained and definite, then the actual Element parameter of Process.**all** shall be unconstrained. 142/2

Ramification: This means that the elements cannot be directly allocated from the heap; it must be possible to change the discriminants of the element in place. 142.a/2

The element at position Index is not an empty element after successful completion of this operation. 143/2

Ramification: Since reading an empty element is a bounded error, attempting to use this procedure to replace empty elements may fail. Use Replace_Element to do that reliably. 143.a/2

```
procedure Update_Element                                                    144/5
  (Container : in out Vector;
   Position  : in     Cursor;
   Process   : not null access procedure
                     (Element : in out Element_Type))
   with Pre  => (Position /= No_Element
                     or else raise Constraint_Error) and then
                (Has_Element (Container, Position)
                     or else raise Program_Error);
```

{*AI95-00302-03*} {*AI05-0264-1*} {*AI05-0265-1*} {*AI12-0112-1*} Update_Element calls Process.**all** with the element designated by Position as the argument. Tampering with the elements of Container is prohibited during the execution of the call on Process.**all**. Any exception raised by Process.**all** is propagated. 145/5

If Element_Type is unconstrained and definite, then the actual Element parameter of Process.**all** shall be unconstrained. 146/2

The element designated by Position is not an empty element after successful completion of this operation. 147/2

```
{AI12-0112-1} type Constant_Reference_Type                                  147.1/5
      (Element : not null access constant Element_Type) is private
   with Implicit_Dereference => Element,
        Nonblocking, Global => in out synchronized,
        Default_Initial_Condition => (raise Program_Error);
```

```
{AI12-0112-1} type Reference_Type (Element : not null access Element_Type) is  147.2/5
private
   with Implicit_Dereference => Element,
        Nonblocking, Global => in out synchronized,
        Default_Initial_Condition => (raise Program_Error);
```

{*AI05-0212-1*} The types Constant_Reference_Type and Reference_Type need finalization. 147.3/3

This paragraph was deleted.{*AI12-0112-1*} 147.4/5

Reason: It is expected that Reference_Type (and Constant_Reference_Type) will be a controlled type, for which finalization will have some action to terminate the tampering check for the associated container. If the object is created 147.a/3

by default, however, there is no associated container. Since this is useless, and supporting this case would take extra work, we define it to raise an exception.

147.5/5
```
{AI12-0112-1} function Constant_Reference (Container : aliased in Vector;
                                           Index     : in Index_Type)
   return Constant_Reference_Type
   with Pre    => Index in First_Index (Container) .. Last_Index (Container)
                     or else raise Constraint_Error,
        Post   => Tampering_With_Cursors_Prohibited (Container),
        Nonblocking, Global => null, Use_Formal => null;
```

147.6/3 {*AI05-0212-1*} {*AI05-0269-1*} This function (combined with the Constant_Indexing and Implicit_Dereference aspects) provides a convenient way to gain read access to an individual element of a vector given an index value.

147.7/5 {*AI05-0212-1*} {*AI05-0265-1*} {*AI12-0112-1*} Constant_Reference returns an object whose discriminant is an access value that designates the element at position Index. Tampering with the elements of Container is prohibited while the object returned by Constant_Reference exists and has not been finalized.

147.8/5
```
{AI12-0112-1} function Reference (Container : aliased in out Vector;
                                  Index     : in Index_Type)
   return Reference_Type
   with Pre    => Index in First_Index (Container) .. Last_Index (Container)
                     or else raise Constraint_Error,
        Post   => Tampering_With_Cursors_Prohibited (Container),
        Nonblocking, Global => null, Use_Formal => null;
```

147.9/3 {*AI05-0212-1*} {*AI05-0269-1*} This function (combined with the Variable_Indexing and Implicit_Dereference aspects) provides a convenient way to gain read and write access to an individual element of a vector given an index value.

147.10/5 {*AI05-0212-1*} {*AI05-0265-1*} {*AI12-0112-1*} Reference returns an object whose discriminant is an access value that designates the element at position Index. Tampering with the elements of Container is prohibited while the object returned by Reference exists and has not been finalized.

147.11/3 The element at position Index is not an empty element after successful completion of this operation.

147.12/5
```
{AI12-0112-1} function Constant_Reference (Container : aliased in Vector;
                                           Position  : in Cursor)
   return Constant_Reference_Type
   with Pre  => (Position /= No_Element
                     or else raise Constraint_Error) and then
                (Has_Element (Container, Position)
                     or else raise Program_Error),
        Post   => Tampering_With_Cursors_Prohibited (Container),
        Nonblocking, Global => null, Use_Formal => null;
```

147.13/3 {*AI05-0212-1*} {*AI05-0269-1*} This function (combined with the Constant_Indexing and Implicit_Dereference aspects) provides a convenient way to gain read access to an individual element of a vector given a cursor.

147.14/5 {*AI05-0212-1*} {*AI05-0265-1*} {*AI12-0112-1*} Constant_Reference returns an object whose discriminant is an access value that designates the element designated by Position. Tampering with the elements of Container is prohibited while the object returned by Constant_Reference exists and has not been finalized.

```
{AI12-0112-1} function Reference (Container : aliased in out Vector;                    147.15/5
                                  Position  : in Cursor)
   return Reference_Type
   with Pre  => (Position /= No_Element
                    or else raise Constraint_Error) and then
                 (Has_Element (Container, Position)
                    or else raise Program_Error),
        Post   => Tampering_With_Cursors_Prohibited (Container),
        Nonblocking, Global => null, Use_Formal => null;
```

{*AI05-0212-1*} {*AI05-0269-1*} This function (combined with the Variable_Indexing and Implicit_Dereference aspects) provides a convenient way to gain read and write access to an individual element of a vector given a cursor. 147.16/3

{*AI05-0212-1*} {*AI05-0265-1*} {*AI12-0112-1*} Reference returns an object whose discriminant is an access value that designates the element designated by Position. Tampering with the elements of Container is prohibited while the object returned by Reference exists and has not been finalized. 147.17/5

The element designated by Position is not an empty element after successful completion of this operation. 147.18/3

```
procedure Assign (Target : in out Vector; Source : in Vector)                           147.19/3
   with Pre  => not Tampering_With_Cursors_Prohibited (Target)
                    or else raise Program_Error,
        Post => Length (Source) = Length (Target) and then
                Capacity (Target) >= Length (Target);
```

{*AI05-0001-1*} {*AI05-0248-1*} {*AI05-0262-1*} If Target denotes the same object as Source, the operation has no effect. If the length of Source is greater than the capacity of Target, Reserve_Capacity (Target, Length (Source)) is called. The elements of Source are then copied to Target as for an assignment_statement assigning Source to Target (this includes setting the length of Target to be that of Source). 147.20/3

Discussion: {*AI05-0005-1*} This routine exists for compatibility with the bounded vector container. For an unbounded vector, `Assign(A, B)` and `A := B` behave identically. For a bounded vector, := will raise an exception if the container capacities are different, while Assign will not raise an exception if there is enough room in the target. 147.b/3

```
function Copy (Source : Vector; Capacity : Count_Type := 0)                             147.21/5
   return Vector
   with Pre => Capacity = 0 or else Capacity >= Length (Source)
                    or else raise Capacity_Error,
        Post => Length (Copy'Result) = Length (Source) and then
                not Tampering_With_Elements_Prohibited (Copy'Result)
                   and then
                not Tampering_With_Cursors_Prohibited (Copy'Result)
                   and then
                Copy'Result.Capacity >= (if Capacity = 0 then
                   Length (Source) else Capacity);
```

{*AI05-0001-1*} {*AI12-0112-1*} Returns a vector whose elements are initialized from the corresponding elements of Source. 147.22/5

148/5
```
procedure Move (Target : in out Vector;
                Source : in out Vector)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Target)
                   or else raise Program_Error) and then
                (not Tampering_With_Cursors_Prohibited (Source)
                   or else raise Program_Error),
        Post => (if not Target'Has_Same_Storage (Source) then
                    Length (Target) = Length (Source)'Old and then
                    Length (Source) = 0 and then
                    Capacity (Target) >= Length (Source)'Old);
```

149/5 {*AI95-00302-03*} {*AI05-0001-1*} {*AI05-0248-1*} {*AI12-0112-1*} If Target denotes the same object as Source, then the operation has no effect. Otherwise, Move first calls Reserve_Capacity (Target, Length (Source)) and then Clear (Target); then, each element from Source is removed from Source and inserted into Target in the original order.

149.a/2 **Discussion:** The idea is that the internal array is removed from Source and moved to Target. (See the Implementation Advice for Move). If Capacity (Target) /= 0, the previous internal array may need to be deallocated. We don't mention this explicitly, because it is covered by the "no memory loss" Implementation Requirement.

150/5
```
procedure Insert_Vector (Container : in out Vector;
                         Before    : in     Extended_Index;
                         New_Item  : in     Vector)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                   or else raise Program_Error) and then
                (Before in
                   First_Index (Container) .. Last_Index (Container) + 1
                   or else raise Constraint_Error) and then
                (Length (Container) <= Maximum_Length - Length (New_Item)
                   or else raise Constraint_Error),
        Post => Length (Container)'Old + Length (New_Item) =
                   Length (Container) and then
                Capacity (Container) >= Length (Container);
```

151/5 {*AI95-00302-03*} {*AI05-0264-1*} {*AI12-0112-1*} {*AI12-0400-1*} If Length(New_Item) is 0, then Insert_Vector does nothing. Otherwise, it computes the new length *NL* as the sum of the current length and Length (New_Item); if the value of Last appropriate for length *NL* would be greater than Index_Type'Last, then Constraint_Error is propagated.

152/5 {*AI12-0400-1*} If the current vector capacity is less than *NL*, Reserve_Capacity (Container, *NL*) is called to increase the vector capacity. Then Insert_Vector slides the elements in the range Before .. Last_Index (Container) up by Length(New_Item) positions, and then copies the elements of New_Item to the positions starting at Before. Any exception raised during the copying is propagated.

152.a/2 **Ramification:** Moving the elements does not necessarily involve copying. Similarly, since Reserve_Capacity does not require the copying of elements, it does not need to be explicitly called (the implementation can combine the operations if it wishes to).

153/5
```
procedure Insert_Vector (Container : in out Vector;
                         Before    : in     Cursor;
                         New_Item  : in     Vector)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                   or else raise Program_Error) and then
                (Before = No_Element or else
                 Has_Element (Container, Before)
                   or else raise Program_Error) and then
                (Length (Container) <= Maximum_Length - Length (New_Item)
                   or else raise Constraint_Error),
        Post => Length (Container)'Old + Length (New_Item) =
                   Length (Container) and then
                Capacity (Container) >= Length (Container);
```

154/5 {*AI95-00302-03*} {*AI05-0264-1*} {*AI12-0112-1*} {*AI12-0400-1*} If Length(New_Item) is 0, then Insert_Vector does nothing. If Before is No_Element, then the call is equivalent to Insert_Vector

(Container, Last_Index (Container) + 1, New_Item); otherwise, the call is equivalent to Insert_Vector (Container, To_Index (Before), New_Item);

Ramification: The check on Before checks that the cursor does not belong to some other Container. This check implies that a reference to the container is included in the cursor value. This wording is not meant to require detection of dangling cursors; such cursors are defined to be invalid, which means that execution is erroneous, and any result is allowed (including not raising an exception). 154.a/2

```
procedure Insert_Vector (Container : in out Vector;                                    155/5
                         Before    : in     Cursor;
                         New_Item  : in     Vector;
                         Position  :    out Cursor)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error) and then
                 (Before = No_Element or else
                  Has_Element (Container, Before)
                    or else raise Program_Error) and then
                 (Length (Container) <= Maximum_Length - Length (New_Item)
                    or else raise Constraint_Error),
        Post => Length (Container)'Old + Length (New_Item) =
                    Length (Container) and then
                 Has_Element (Container, Position) and then
                 Capacity (Container) >= Length (Container);
```

{*AI95-00302-03*} {*AI12-0112-1*} {*AI12-0400-1*} If Before equals No_Element, then let *T* be Last_Index (Container) + 1; otherwise, let *T* be To_Index (Before). Insert_Vector (Container, *T*, New_Item) is called, and then Position is set to To_Cursor (Container, *T*). 156/5

Discussion: {*AI12-0400-1*} The messy wording is needed because Before is invalidated by Insert_Vector, and we don't want Position to be invalid after this call. An implementation probably only needs to copy Before to Position. 156.a/5

```
{AI12-0112-1} procedure Insert (Container : in out Vector;                             157/5
                   Before    : in     Extended_Index;
                   New_Item  : in     Element_Type;
                   Count     : in     Count_Type := 1)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error) and then
                 (Before in
                    First_Index (Container) .. Last_Index (Container) + 1
                    or else raise Constraint_Error) and then
                 (Length (Container) <= Maximum_Length - Count
                    or else raise Constraint_Error),
        Post => Length (Container)'Old + Count = Length (Container) and then
                 Capacity (Container) >= Length (Container);
```

{*AI95-00302-03*} Equivalent to Insert (Container, Before, To_Vector (New_Item, Count)); 158/2

```
{AI12-0112-1} procedure Insert (Container : in out Vector;                             159/5
                   Before    : in     Cursor;
                   New_Item  : in     Element_Type;
                   Count     : in     Count_Type := 1)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error) and then
                 (Before = No_Element or else
                  Has_Element (Container, Before)
                    or else raise Program_Error) and then
                 (Length (Container) <= Maximum_Length - Count
                    or else raise Constraint_Error),
        Post => Length (Container)'Old + Count = Length (Container) and then
                 Capacity (Container) >= Length (Container);
```

{*AI95-00302-03*} Equivalent to Insert (Container, Before, To_Vector (New_Item, Count)); 160/2

161/5
```
{AI12-0112-1} procedure Insert (Container : in out Vector;
                   Before    : in     Cursor;
                   New_Item  : in     Element_Type;
                   Position  :    out Cursor;
                   Count     : in     Count_Type := 1)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                   or else raise Program_Error) and then
                (Before = No_Element or else
                 Has_Element (Container, Before)
                   or else raise Program_Error) and then
                (Length (Container) <= Maximum_Length - Count
                   or else raise Constraint_Error),
        Post => Length (Container)'Old + Count = Length (Container) and then
                Has_Element (Container, Position) and then
                Capacity (Container) >= Length (Container);
```

162/2 {*AI95-00302-03*} Equivalent to Insert (Container, Before, To_Vector (New_Item, Count), Position);

162.a/3 **Ramification:** {*AI05-0257-1*} If Count equals 0, Position will designate the element designated by Before, rather than a newly inserted element. Otherwise, Position will designate the first newly inserted element.

163/5
```
procedure Insert (Container : in out Vector;
                  Before    : in     Extended_Index;
                  Count     : in     Count_Type := 1)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                   or else raise Program_Error) and then
                (Before in
                   First_Index (Container) .. Last_Index (Container) + 1
                   or else raise Constraint_Error) and then
                (Length (Container) <= Maximum_Length - Count
                   or else raise Constraint_Error),
        Post => Length (Container)'Old + Count = Length (Container) and then
                Capacity (Container) >= Length (Container);
```

164/5 {*AI95-00302-03*} {*AI05-0264-1*} {*AI12-0112-1*} If Count is 0, then Insert does nothing. Otherwise, it computes the new length *NL* as the sum of the current length and Count; if the value of Last appropriate for length *NL* would be greater than Index_Type'Last, then Constraint_Error is propagated.

165/2 If the current vector capacity is less than *NL*, Reserve_Capacity (Container, *NL*) is called to increase the vector capacity. Then Insert slides the elements in the range Before .. Last_Index (Container) up by Count positions, and then inserts elements that are initialized by default (see 3.3.1) in the positions starting at Before.

166/5
```
procedure Insert (Container : in out Vector;
                  Before    : in     Cursor;
                  Position  :    out Cursor;
                  Count     : in     Count_Type := 1)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                   or else raise Program_Error) and then
                (Before = No_Element or else
                 Has_Element (Container, Before)
                   or else raise Program_Error) and then
                (Length (Container) <= Maximum_Length - Count
                   or else raise Constraint_Error),
        Post => Length (Container)'Old + Count = Length (Container) and then
                Has_Element (Container, Position) and then
                Capacity (Container) >= Length (Container);
```

167/5 {*AI95-00302-03*} {*AI12-0112-1*} If Before equals No_Element, then let *T* be Last_Index (Container) + 1; otherwise, let *T* be To_Index (Before). Insert (Container, *T*, Count) is called, and then Position is set to To_Cursor (Container, *T*).

Reason: This routine exists mainly to ease conversion between Vector and List containers. Unlike Insert_Space, this routine default initializes the elements it inserts, which can be more expensive for some element types. 167.a/2

```
{AI12-0080-1} {AI12-0112-1} {AI12-0400-1} procedure Prepend_Vector (Container : in out Vector;
                          New_Item  : in     Vector)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error) and then
                 (Length (Container) <= Maximum_Length - Length (New_Item)
                    or else raise Constraint_Error),
        Post => Length (Container)'Old + Length (New_Item) =
                    Length (Container) and then
                 Capacity (Container) >= Length (Container);
```
168/5

{*AI95-00302-03*} Equivalent to Insert (Container, First_Index (Container), New_Item). 169/2

```
{AI12-0112-1} procedure Prepend (Container : in out Vector;
                    New_Item  : in     Element_Type;
                    Count     : in     Count_Type := 1)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error) and then
                 (Length (Container) <= Maximum_Length - Count
                    or else raise Constraint_Error),
        Post => Length (Container)'Old + Count = Length (Container) and then
                 Capacity (Container) >= Length (Container);
```
170/5

{*AI95-00302-03*} Equivalent to Insert (Container, First_Index (Container), New_Item, Count). 171/2

```
{AI12-0112-1} {AI12-0400-1} procedure Append_Vector (Container : in out Vector;
                         New_Item  : in     Vector)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error) and then
                 (Length (Container) <= Maximum_Length - Length (New_Item)
                    or else raise Constraint_Error),
        Post => Length (Container)'Old + Length (New_Item) =
                    Length (Container) and then
                 Capacity (Container) >= Length (Container);
```
172/5

{*AI95-00302-03*} Equivalent to Insert (Container, Last_Index (Container) + 1, New_Item). 173/2

```
{AI12-0112-1} {AI12-0400-1} procedure Append (Container : in out Vector;
                      New_Item  : in     Element_Type;
                      Count     : in     Count_Type)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error) and then
                 (Length (Container) <= Maximum_Length - Count
                    or else raise Constraint_Error),
        Post => Length (Container)'Old + Count = Length (Container) and then
                 Capacity (Container) >= Length (Container);
```
174/5

{*AI95-00302-03*} Equivalent to Insert (Container, Last_Index (Container) + 1, New_Item, Count). 175/2

```
procedure Append (Container : in out Vector;
                      New_Item  : in     Element_Type)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error) and then
                 (Length (Container) <= Maximum_Length - 1
                    or else raise Constraint_Error),
        Post => Length (Container)'Old + 1 = Length (Container) and then
                 Capacity (Container) >= Length (Container);
```
175.1/5

{*AI12-0112-1*} {*AI12-0212-1*} {*AI12-0400-1*} Equivalent to Insert (Container, Last_Index (Container) + 1, New_Item, 1). 175.2/5

176/5
```
procedure Insert_Space (Container : in out Vector;
                        Before    : in     Extended_Index;
                        Count     : in     Count_Type := 1)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                   or else raise Program_Error) and then
                (Before in
                   First_Index (Container) .. Last_Index (Container) + 1
                   or else raise Constraint_Error) and then
                (Length (Container) <= Maximum_Length - Count
                   or else raise Constraint_Error),
        Post => Length (Container)'Old + Count = Length (Container) and then
                Capacity (Container) >= Length (Container);
```

177/5 {*AI95-00302-03*} {*AI05-0264-1*} {*AI12-0112-1*} If Count is 0, then Insert_Space does nothing. Otherwise, it computes the new length *NL* as the sum of the current length and Count; if the value of Last appropriate for length *NL* would be greater than Index_Type'Last, then Constraint_Error is propagated.

178/2 If the current vector capacity is less than *NL*, Reserve_Capacity (Container, *NL*) is called to increase the vector capacity. Then Insert_Space slides the elements in the range Before .. Last_Index (Container) up by Count positions, and then inserts empty elements in the positions starting at Before.

179/5
```
procedure Insert_Space (Container : in out Vector;
                        Before    : in     Cursor;
                        Position  :    out Cursor;
                        Count     : in     Count_Type := 1)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                   or else raise Program_Error) and then
                (Before = No_Element or else
                   Has_Element (Container, Before)
                   or else raise Program_Error) and then
                (Length (Container) <= Maximum_Length - Count
                   or else raise Constraint_Error),
        Post => Length (Container)'Old + Count = Length (Container) and then
                Has_Element (Container, Position) and then
                Capacity (Container) >= Length (Container);
```

180/5 {*AI95-00302-03*} {*AI12-0112-1*} If Before equals No_Element, then let *T* be Last_Index (Container) + 1; otherwise, let *T* be To_Index (Before). Insert_Space (Container, *T*, Count) is called, and then Position is set to To_Cursor (Container, *T*).

181/5
```
procedure Delete (Container : in out Vector;
                  Index     : in     Extended_Index;
                  Count     : in     Count_Type := 1)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                   or else raise Program_Error) and then
                (Index in
                   First_Index (Container) .. Last_Index (Container) + 1
                   or else raise Constraint_Error),
        Post => Length (Container)'Old - Count <= Length (Container);
```

182/5 {*AI95-00302-03*} {*AI05-0264-1*} {*AI12-0112-1*} If Count is 0, Delete has no effect. Otherwise, Delete slides the elements (if any) starting at position Index + Count down to Index. Any exception raised during element assignment is propagated.

182.a/2 **Ramification:** If Index + Count >= Last_Index(Container), this effectively truncates the vector (setting Last_Index to Index – 1 and consequently sets Length to Index – Index_Type'First).

```
procedure Delete (Container : in out Vector;                                    183/5
                  Position  : in out Cursor;
                  Count     : in     Count_Type := 1)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error) and then
                 (Position /= No_Element
                    or else raise Constraint_Error) and then
                 (Has_Element (Container, Position)
                    or else raise Program_Error),
        Post => Length (Container)'Old - Count <= Length (Container)
                and then Position = No_Element;
```

{*AI95-00302-03*} {*AI12-0112-1*} Delete (Container, To_Index (Position), Count) is called, and then Position is set to No_Element. 184/5

```
{AI12-0112-1} procedure Delete_First (Container : in out Vector;                185/5
                      Count     : in     Count_Type := 1)
   with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error,
        Post => Length (Container)'Old - Count <= Length (Container);
```

{*AI95-00302-03*} Equivalent to Delete (Container, First_Index (Container), Count). 186/2

```
{AI12-0112-1} procedure Delete_Last (Container : in out Vector;                 187/5
                     Count     : in     Count_Type := 1)
   with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error,
        Post => Length (Container)'Old - Count <= Length (Container);
```

{*AI95-00302-03*} {*AI05-0264-1*} If Length (Container) <= Count, then Delete_Last is equivalent to Clear (Container). Otherwise, it is equivalent to Delete (Container, Index_Type'Val(Index_Type'Pos(Last_Index (Container)) – Count + 1), Count). 188/3

```
{AI05-0092-1} {AI12-0112-1} procedure Reverse_Elements (Container : in out Vector)  189/5
   with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error;
```

{*AI95-00302-03*} Reorders the elements of Container in reverse order. 190/2

Discussion: This can copy the elements of the vector — all cursors referencing the vector are ambiguous afterwards and may designate different elements afterwards. 190.a/2

```
procedure Swap (Container : in out Vector;                                      191/5
                I, J      : in     Index_Type)
   with Pre => (not Tampering_With_Elements_Prohibited (Container)
                   or else raise Program_Error) and then
               (I in First_Index (Container) .. Last_Index (Container)
                   or else raise Constraint_Error) and then
               (J in First_Index (Container) .. Last_Index (Container)
                   or else raise Constraint_Error);
```

{*AI95-00302-03*} {*AI12-0112-1*} Swap exchanges the values of the elements at positions I and J. 192/5

To be honest: The implementation is not required to actually copy the elements if it can do the swap some other way. But it is allowed to copy the elements if needed. 192.a/2

193/5
```
procedure Swap (Container : in out Vector;
                I, J      : in     Cursor)
   with Pre  => (not Tampering_With_Elements_Prohibited (Container)
                   or else raise Program_Error) and then
                (I /= No_Element or else Constraint_Error) and then
                (J /= No_Element or else Constraint_Error) and then
                (Has_Element (Container, I)
                   or else raise Program_Error) and then
                (Has_Element (Container, J)
                   or else raise Program_Error);
```

194/5 {*AI95-00302-03*} {*AI12-0112-1*} Swap exchanges the values of the elements designated by I and J.

194.a/2 **Ramification:** After a call to Swap, I designates the element value previously designated by J, and J designates the element value previously designated by I. The cursors do not become ambiguous from this operation.

194.b/2 **To be honest:** The implementation is not required to actually copy the elements if it can do the swap some other way. But it is allowed to copy the elements if needed.

195/5
```
{AI12-0112-1} function First_Index (Container : Vector) return Index_Type
   with Nonblocking, Global => null, Use_Formal => null,
        Post => First_Index'Result = Index_Type'First;
```

196/2 {*AI95-00302-03*} Returns the value Index_Type'First.

196.a/2 **Discussion:** We'd rather call this “First”, but then calling most routines in here with First (Some_Vect) would be ambiguous.

197/5
```
{AI12-0112-1} function First (Container : Vector) return Cursor
   with Nonblocking, Global => null, Use_Formal => null,
        Post => (if not Is_Empty (Container)
                 then Has_Element (Container, First'Result)
                 else First'Result = No_Element);
```

198/2 {*AI95-00302-03*} If Container is empty, First returns No_Element. Otherwise, it returns a cursor that designates the first element in Container.

199/5
```
{AI12-0112-1} function First_Element (Container : Vector)
   return Element_Type
   with Pre => (not Is_Empty (Container)
                   or else raise Constraint_Error);
```

200/2 {*AI95-00302-03*} Equivalent to Element (Container, First_Index (Container)).

201/5
```
{AI12-0112-1} function Last_Index (Container : Vector) return Extended_Index
   with Nonblocking, Global => null, Use_Formal => null,
        Post => (if Length (Container) = 0 then Last_Index'Result = No_Index
                 else Count_Type(Last_Index'Result - Index_Type'First) =
                      Length (Container) - 1);
```

202/2 {*AI95-00302-03*} If Container is empty, Last_Index returns No_Index. Otherwise, it returns the position of the last element in Container.

203/5
```
{AI12-0112-1} function Last (Container : Vector) return Cursor
   with Nonblocking, Global => null, Use_Formal => null,
        Post => (if not Is_Empty (Container)
                 then Has_Element (Container, Last'Result)
                 else Last'Result = No_Element);
```

204/2 {*AI95-00302-03*} If Container is empty, Last returns No_Element. Otherwise, it returns a cursor that designates the last element in Container.

205/5
```
{AI12-0112-1} function Last_Element (Container : Vector)
   return Element_Type
   with Pre => (not Is_Empty (Container)
                   or else raise Constraint_Error);
```

206/2 {*AI95-00302-03*} Equivalent to Element (Container, Last_Index (Container)).

```
{AI12-0112-1} function Next (Position : Cursor) return Cursor                    207/5
   with Nonblocking, Global => in all, Use_Formal => null,
        Post => (if Position = No_Element then Next'Result = No_Element);
```

{*AI95-00302-03*} If Position equals No_Element or designates the last element of the container, then Next returns the value No_Element. Otherwise, it returns a cursor that designates the element with index To_Index (Position) + 1 in the same vector as Position. 208/2

```
function Next (Container : Vector;                                               208.1/5
               Position : Cursor) return Cursor
   with Nonblocking, Global => null, Use_Formal => null,
        Pre  => Position = No_Element or else
                Has_Element (Container, Position)
                   or else raise Program_Error,
        Post => (if Position = No_Element then Next'Result = No_Element
                 elsif Has_Element (Container, Next'Result) then
                    To_Index (Container, Next'Result) =
                    To_Index (Container, Position) + 1
                 elsif Next'Result = No_Element then
                    Position = Last (Container)
                 else False);
```

{*AI12-0112-1*} Returns a cursor designating the next element in Container, if any. 208.2/5

```
{AI12-0112-1} procedure Next (Position : in out Cursor)                          209/5
   with Nonblocking, Global => in all, Use_Formal => null;
```

{*AI95-00302-03*} Equivalent to Position := Next (Position). 210/2

```
procedure Next (Container : in     Vector;                                       210.1/5
                Position  : in out Cursor)
   with Nonblocking, Global => null, Use_Formal => null,
        Pre  => Position = No_Element or else
                Has_Element (Container, Position)
                   or else raise Program_Error,
        Post => (if Position /= No_Element
                 then Has_Element (Container, Position));
```

{*AI12-0112-1*} Equivalent to Position := Next (Container, Position). 210.2/5

```
{AI12-0112-1} function Previous (Position : Cursor) return Cursor                211/5
   with Nonblocking, Global => in all, Use_Formal => null,
        Post => (if Position = No_Element
                 then Previous'Result = No_Element);
```

{*AI95-00302-03*} If Position equals No_Element or designates the first element of the container, then Previous returns the value No_Element. Otherwise, it returns a cursor that designates the element with index To_Index (Position) – 1 in the same vector as Position. 212/2

```
function Previous (Container : Vector;                                           212.1/5
                   Position  : Cursor) return Cursor
   with Nonblocking, Global => null, Use_Formal => null,
        Pre  => Position = No_Element or else
                Has_Element (Container, Position)
                   or else raise Program_Error,
        Post => (if Position = No_Element then Previous'Result = No_Element
                 elsif Has_Element (Container, Previous'Result) then
                   To_Index (Container, Previous'Result) =
                   To_Index (Container, Position) - 1
                 elsif Previous'Result = No_Element then
                   Position = First (Container)
                 else False);
```

{*AI12-0112-1*} Returns a cursor designating the previous element in Container, if any. 212.2/5

213/5
```
{AI12-0112-1} procedure Previous (Position : in out Cursor)
   with Nonblocking, Global => in all, Use_Formal => null;
```

214/2 {*AI95-00302-03*} Equivalent to Position := Previous (Position).

214.1/5
```
procedure Previous (Container : in     Vector;
                    Position  : in out Cursor)
   with Nonblocking, Global => null, Use_Formal => null,
        Pre  => Position = No_Element or else
                Has_Element (Container, Position)
                   or else raise Program_Error,
        Post => (if Position /= No_Element
                 then Has_Element (Container, Position));
```

214.2/5 {*AI12-0112-1*} Equivalent to Position := Previous (Container, Position).

215/2
```
function Find_Index (Container : Vector;
                     Item      : Element_Type;
                     Index     : Index_Type := Index_Type'First)
   return Extended_Index;
```

216/2 {*AI95-00302-03*} Searches the elements of Container for an element equal to Item (using the generic formal equality operator). The search starts at position Index and proceeds towards Last_Index (Container). If no equal element is found, then Find_Index returns No_Index. Otherwise, it returns the index of the first equal element encountered.

217/5
```
function Find (Container : Vector;
               Item      : Element_Type;
               Position  : Cursor := No_Element)
   return Cursor
   with Pre  => Position = No_Element or else
                Has_Element (Container, Position)
                   or else raise Program_Error,
        Post => (if Find'Result /= No_Element
                 then Has_Element (Container, Find'Result));
```

218/5 {*AI95-00302-03*} {*AI05-0264-1*} {*AI12-0112-1*} Find searches the elements of Container for an element equal to Item (using the generic formal equality operator). The search starts at the first element if Position equals No_Element, and at the element designated by Position otherwise. It proceeds towards the last element of Container. If no equal element is found, then Find returns No_Element. Otherwise, it returns a cursor designating the first equal element encountered.

219/2
```
function Reverse_Find_Index (Container : Vector;
                             Item      : Element_Type;
                             Index     : Index_Type := Index_Type'Last)
   return Extended_Index;
```

220/2 {*AI95-00302-03*} Searches the elements of Container for an element equal to Item (using the generic formal equality operator). The search starts at position Index or, if Index is greater than Last_Index (Container), at position Last_Index (Container). It proceeds towards First_Index (Container). If no equal element is found, then Reverse_Find_Index returns No_Index. Otherwise, it returns the index of the first equal element encountered.

```
function Reverse_Find (Container : Vector;                                    221/5
                       Item      : Element_Type;
                       Position  : Cursor := No_Element)
   return Cursor
   with Pre  => Position = No_Element or else
                Has_Element (Container, Position)
                   or else raise Program_Error,
        Post => (if Reverse_Find'Result /= No_Element
                 then Has_Element (Container, Reverse_Find'Result));
```

{*AI95-00302-03*} {*AI05-0264-1*} {*AI12-0112-1*} Reverse_Find searches the elements of Container for an element equal to Item (using the generic formal equality operator). The search starts at the last element if Position equals No_Element, and at the element designated by Position otherwise. It proceeds towards the first element of Container. If no equal element is found, then Reverse_Find returns No_Element. Otherwise, it returns a cursor designating the first equal element encountered. 222/5

```
function Contains (Container : Vector;                                        223/2
                   Item      : Element_Type) return Boolean;
```

{*AI95-00302-03*} Equivalent to Has_Element (Find (Container, Item)). 224/2

Paragraphs 225 and 226 were moved above.

```
{AI12-0112-1} procedure Iterate                                               227/5
  (Container : in Vector;
   Process   : not null access procedure (Position : in Cursor))
   with Allows_Exit;
```

{*AI95-00302-03*} {*AI05-0265-1*} Invokes Process.**all** with a cursor that designates each element in Container, in index order. Tampering with the cursors of Container is prohibited during the execution of a call on Process.**all**. Any exception raised by Process.**all** is propagated. 228/3

Discussion: The purpose of the "tamper with the cursors" check is to prevent erroneous execution from the Position parameter of Process.**all** becoming invalid. This check takes place when the operations that tamper with the cursors of the container are called. The check cannot be made later (say in the body of Iterate), because that could cause the Position cursor to be invalid and potentially cause execution to become erroneous -- defeating the purpose of the check. 228.a/2

There is no check needed if an attempt is made to insert or delete nothing (that is, Count = 0 or Length(Item) = 0). 228.b/2

The check is easy to implement: each container needs a counter. The counter is incremented when Iterate is called, and decremented when Iterate completes. If the counter is nonzero when an operation that inserts or deletes is called, Finalize is called, or one of the other operations in the list occurs, Program_Error is raised. 228.c/2

```
{AI12-0112-1} procedure Reverse_Iterate                                       229/5
  (Container : in Vector;
   Process   : not null access procedure (Position : in Cursor))
   with Allows_Exit;
```

{*AI95-00302-03*} {*AI05-0212-1*} Iterates over the elements in Container as per procedure Iterate, except that elements are traversed in reverse index order. 230/3

```
{AI12-0212-1} {AI12-0266-1} function Iterate (Container : in Vector)          230.1/5
   return Vector_Iterator_Interfaces.Parallel_Reversible_Iterator'Class
   with Post   => Tampering_With_Cursors_Prohibited (Container);
```

{*AI05-0212-1*} {*AI05-0265-1*} {*AI05-0269-1*} {*AI12-0266-1*} Iterate returns an iterator object (see 5.5.1) that will generate a value for a loop parameter (see 5.5.2) designating each node in Container, starting with the first node and moving the cursor as per the Next function when used as a forward iterator, and starting with the last node and moving the cursor as per the Previous function when used as a reverse iterator, and processing all nodes concurrently when used as a parallel iterator. Tampering with the cursors of Container is prohibited while the iterator object 230.2/5

exists (in particular, in the sequence_of_statements of the loop_statement whose iterator_specification denotes this object). The iterator object needs finalization.

230.3/5
```
function Iterate (Container : in Vector; Start : in Cursor)
   return Vector_Iterator_Interfaces.Reversible_Iterator'Class
   with Pre    => (Start /= No_Element
                      or else raise Constraint_Error) and then
                  (Has_Element (Container, Start)
                      or else raise Program_Error),
        Post   => Tampering_With_Cursors_Prohibited (Container);
```

230.4/5 {*AI05-0212-1*} {*AI05-0262-1*} {*AI05-0265-1*} {*AI05-0269-1*} {*AI12-0212-1*} Iterate returns a reversible iterator object (see 5.5.1) that will generate a value for a loop parameter (see 5.5.2) designating each node in Container, starting with the node designated by Start and moving the cursor as per the Next function when used as a forward iterator, or moving the cursor as per the Previous function when used as a reverse iterator. Tampering with the cursors of Container is prohibited while the iterator object exists (in particular, in the sequence_of_statements of the loop_statement whose iterator_specification denotes this object). The iterator object needs finalization.

230.a/3 **Discussion:** Exits are allowed from the loops created using the iterator objects. In particular, to stop the iteration at a particular cursor, just add

230.b/3
```
exit when Cur = Stop;
```

230.c/3 in the body of the loop (assuming that `Cur` is the loop parameter and `Stop` is the cursor that you want to stop at).

231/3 {*AI05-0044-1*} {*AI05-0262-1*} The actual function for the generic formal function "<" of Generic_Sorting is expected to return the same value each time it is called with a particular pair of element values. It should define a strict weak ordering relationship (see A.18); it should not modify Container. If the actual for "<" behaves in some other manner, the behavior of the subprograms of Generic_Sorting are unspecified. The number of times the subprograms of Generic_Sorting call "<" is unspecified.

232/2
```
function Is_Sorted (Container : Vector) return Boolean;
```

233/2 {*AI95-00302-03*} Returns True if the elements are sorted smallest first as determined by the generic formal "<" operator; otherwise, Is_Sorted returns False. Any exception raised during evaluation of "<" is propagated.

234/5
```
{AI12-0112-1} procedure Sort (Container : in out Vector)
   with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error;
```

235/2 {*AI95-00302-03*} Reorders the elements of Container such that the elements are sorted smallest first as determined by the generic formal "<" operator provided. Any exception raised during evaluation of "<" is propagated.

235.a/2 **Ramification:** This implies swapping the elements, usually including an intermediate copy. This means that the elements will usually be copied. (As with Swap, if the implementation can do this some other way, it is allowed to.) Since the elements are nonlimited, this usually will not be a problem. Note that there is Implementation Advice below that the implementation should use a sort that minimizes copying of elements.

235.b/2 The sort is not required to be stable (and the fast algorithm required will not be stable). If a stable sort is needed, the user can include the original location of the element as an extra "sort key". We considered requiring the implementation to do that, but it is mostly extra overhead -- usually there is something already in the element that provides the needed stability.

```
procedure Merge (Target  : in out Vector;                                    236/5
                 Source  : in out Vector)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Target)
                    or else raise Program_Error) and then
                 (not Tampering_With_Cursors_Prohibited (Source)
                    or else raise Program_Error) and then
                 (Length (Target) <= Maximum_Length - Length (Source)
                    or else raise Constraint_Error) and then
                 ((Length (Source) = 0 or else
                    not Target'Has_Same_Storage (Source))
                    or else raise Program_Error),
        Post => (declare
                     Result_Length : constant Count_Type :=
                        Length (Source)'Old + Length (Target)'Old;
                  begin
                     (Length (Source) = 0 and then
                      Length (Target) = Result_Length and then
                      Capacity (Target) >= Result_Length));
```

{*AI95-00302-03*} {*AI05-0021-1*} {*AI12-0112-1*} Merge removes elements from Source and inserts them into Target; afterwards, Target contains the union of the elements that were initially in Source and Target; Source is left empty. If Target and Source are initially sorted smallest first, then Target is ordered smallest first as determined by the generic formal "<" operator; otherwise, the order of elements in Target is unspecified. Any exception raised during evaluation of "<" is propagated. 237/5

Discussion: It is a bounded error if either of the vectors is unsorted, see below. The bounded error can be recovered by sorting Target after the merge call, or the vectors can be pretested with Is_Sorted. 237.a/2

Implementation Note: The Merge operation will usually require copying almost all of the elements. One implementation strategy would be to extend Target to the appropriate length, then copying elements from the back of the vectors working towards the front. An alternative approach would be to allocate a new internal data array of the appropriate length, copy the elements into it in an appropriate order, and then replacing the data array in Target with the temporary. 237.b/2

{*AI12-0111-1*} The nested package Vectors.Stable provides a type Stable.Vector that represents a *stable* vector, which is one that cannot grow and shrink. Such a vector can be created by calling the To_Vector or Copy functions, or by establishing a *stabilized view* of an ordinary vector. 237.1/5

{*AI12-0111-1*} The subprograms of package Containers.Vectors that have a parameter or result of type Vector are included in the nested package Stable with the same specification, except that the following are omitted: 237.2/5

{*AI12-0111-1*} {*AI12-0400-1*} Tampering_With_Cursors_Prohibited, Tampering_With_Elements_Prohibited, Reserve_Capacity, Assign, Move, Insert, Insert_Space, Insert_Vector, Append, Append_Vector, Prepend, Prepend_Vector, Clear, Delete, Delete_First, Delete_Last, and Set_Length 237.3/5

The generic package Generic_Sorting is also included with the same specification, except that Merge is omitted. 237.4/5

Ramification: The names Vector and Cursor mean the types declared in the nested package in these subprogram specifications. 237.b.1/5

Reason: The omitted routines are those that tamper with cursors or elements (or test that state). The model is that it is impossible to tamper with cursors or elements of a stable view since no such operations are included. Thus tampering checks are not needed for a stable view, and we omit the operations associated with those checks. 237.b.2/5

{*AI12-0111-1*} The operations of this package are equivalent to those for ordinary vectors, except that the calls to Tampering_With_Cursors_Prohibited and Tampering_With_Elements_Prohibited that occur in preconditions are replaced by False, and any that occur in postconditions are replaced by True. 237.5/5

237.6/5 {*AI12-0111-1*} {*AI12-0439-1*} If a stable vector is declared with the Base discriminant designating a pre-existing ordinary vector, the stable vector represents a stabilized view of the underlying ordinary vector, and any operation on the stable vector is reflected on the underlying ordinary vector. While a stabilized view exists, any operation that tampers with elements performed on the underlying vector is prohibited. The finalization of a stable vector that provides such a view removes this restriction on the underlying ordinary vector [(though some other restriction can exist due to other concurrent iterations or stabilized views)].

237.7/5 {*AI12-0111-1*} {*AI12-0438-1*} If a stable vector is declared without specifying Base, the object is necessarily initialized. The initializing expression of the stable vector, [typically a call on To_Vector or Copy], determines the Length of the vector. The Length of a stable vector never changes after initialization.

237.c/5 **Proof:** {*AI12-0438-1*} Initialization is required as the type is indefinite, see 3.3.1.

Bounded (Run-Time) Errors

238/3 {*AI95-00302-03*} {*AI05-0212-1*} Reading the value of an empty element by calling Element, Query_Element, Update_Element, Constant_Reference, Reference, Swap, Is_Sorted, Sort, Merge, "=", Find, or Reverse_Find is a bounded error. The implementation may treat the element as having any normal value (see 13.9.1) of the element type, or raise Constraint_Error or Program_Error before modifying the vector.

238.a/2 **Ramification:** For instance, a default initialized element could be returned. Or some previous value of an element. But returning random junk is not allowed if the type has default initial value(s).

238.b/2 Assignment and streaming of empty elements are **not** bounded errors. This is consistent with regular composite types, for which assignment and streaming of uninitialized components do not cause a bounded error, but reading the uninitialized component does cause a bounded error.

238.c/2 There are other operations which are defined in terms of the operations listed above.

239/2 {*AI95-00302-03*} Calling Merge in an instance of Generic_Sorting with either Source or Target not ordered smallest first using the provided generic formal "<" operator is a bounded error. Either Program_Error is raised after Target is updated as described for Merge, or the operation works as defined.

239.1/3 {*AI05-0022-1*} {*AI05-0248-1*} It is a bounded error for the actual function associated with a generic formal subprogram, when called as part of an operation of this package, to tamper with elements of any Vector parameter of the operation. Either Program_Error is raised, or the operation works as defined on the value of the Vector either prior to, or subsequent to, some or all of the modifications to the Vector.

239.2/3 {*AI05-0027-1*} It is a bounded error to call any subprogram declared in the visible part of Containers.Vectors when the associated container has been finalized. If the operation takes Container as an **in out** parameter, then it raises Constraint_Error or Program_Error. Otherwise, the operation either proceeds as it would for an empty container, or it raises Constraint_Error or Program_Error.

240/2 {*AI95-00302-03*} A Cursor value is *ambiguous* if any of the following have occurred since it was created:

241/5 - {*AI12-0400-1*} Insert, Insert_Space, Insert_Vector, or Delete has been called on the vector that contains the element the cursor designates with an index value (or a cursor designating an element at such an index value) less than or equal to the index value of the element designated by the cursor; or

242/2 - The vector that contains the element it designates has been passed to the Sort or Merge procedures of an instance of Generic_Sorting, or to the Reverse_Elements procedure.

243/2 {*AI95-00302-03*} It is a bounded error to call any subprogram other than "=" or Has_Element declared in Containers.Vectors with an ambiguous (but not invalid, see below) cursor parameter. Possible results are:

244/2 - The cursor may be treated as if it were No_Element;

- The cursor may designate some element in the vector (but not necessarily the element that it originally designated); 245/2
- Constraint_Error may be raised; or 246/2
- Program_Error may be raised. 247/2

Reason: Cursors are made ambiguous if an Insert or Delete occurs that moves the elements in the internal array including the designated ones. After such an operation, the cursor probably still designates an element (although it might not after a deletion), but it is a *different* element. That violates the definition of cursor — it designates a particular element. 247.a/2

For "=" or Has_Element, the cursor works normally (it would not be No_Element). We don't want to trigger an exception simply for comparing a bad cursor. 247.b/2

While it is possible to check for these cases or ensure that cursors survive such operations, in many cases the overhead necessary to make the check (or ensure cursors continue to designate the same element) is substantial in time or space. 247.c/2

Erroneous Execution

{*AI95-00302-03*} A Cursor value is *invalid* if any of the following have occurred since it was created: 248/2

- The vector that contains the element it designates has been finalized; 249/2
- {*AI05-0160-1*} The vector that contains the element it designates has been used as the Target of a call to Assign, or as the target of an assignment_statement; 249.1/3
- [The vector that contains the element it designates has been used as the Source or Target of a call to Move;] or 250/2

Proof: {*AI05-0001-1*} Move has been reworded in terms of Assign and Clear, which are covered by other bullets, so this text is redundant. 250.a/3

- {*AI05-0160-1*} {*AI05-0262-1*} The element it designates has been deleted or removed from the vector that previously contained the element. 251/3

Ramification: {*AI05-0160-1*} An element can be removed via calls to Set_Length, Clear, and Merge; and indirectly via calls to Assign and Move. 251.a/3

{*AI95-00302-03*} The result of "=" or Has_Element is unspecified if it is called with an invalid cursor parameter. Execution is erroneous if any other subprogram declared in Containers.Vectors is called with an invalid cursor parameter. 252/2

Discussion: The list above (combined with the bounded error cases) is intended to be exhaustive. In other cases, a cursor value continues to designate its original element. For instance, cursor values survive the appending of new elements. 252.a/2

{*AI05-0212-1*} Execution is erroneous if the vector associated with the result of a call to Reference or Constant_Reference is finalized before the result object returned by the call to Reference or Constant_Reference is finalized. 252.1/3

Reason: Each object of Reference_Type and Constant_Reference_Type probably contains some reference to the originating container. If that container is prematurely finalized (which is only possible via Unchecked_Deallocation, as accessibility checks prevent passing a container to Reference that will not live as long as the result), the finalization of the object of Reference_Type will try to access a nonexistent object. This is a normal case of a dangling pointer created by Unchecked_Deallocation; we have to explicitly mention it here as the pointer in question is not visible in the specification of the type. (This is the same reason we have to say this for invalid cursors.) 252.b/3

Implementation Requirements

{*AI95-00302-03*} No storage associated with a vector object shall be lost upon assignment or scope exit. 253/2

{*AI95-00302-03*} {*AI05-0262-1*} The execution of an assignment_statement for a vector shall have the effect of copying the elements from the source vector object to the target vector object and changing the length of the target object to that of the source object. 254/3

Implementation Note: {*AI05-0298-1*} {*AI12-0005-1*} An assignment of a Vector is a “deep” copy; that is the elements are copied as well as the data structures. We say “effect of” in order to allow the implementation to avoid 254.a/5

copying elements immediately if it wishes. For instance, an implementation that avoided copying until one of the containers is modified would be allowed. (Note that such an implementation would require care, as Query_Element and Constant_Reference both could be used to access an element which later needs to be reallocated while the parameter or reference still exists, potentially leaving the parameter or reference pointing at the wrong element.)

Implementation Advice

255/2 {*AI95-00302-03*} Containers.Vectors should be implemented similarly to an array. In particular, if the length of a vector is *N*, then

256/2 • the worst-case time complexity of Element should be *O*(log *N*);

256.a/2 **Implementation Advice:** The worst-case time complexity of Element for Containers.Vector should be *O*(log *N*).

257/2 • the worst-case time complexity of Append with Count=1 when *N* is less than the capacity of the vector should be *O*(log *N*); and

257.a/2 **Implementation Advice:** The worst-case time complexity of Append with Count = 1 when *N* is less than the capacity for Containers.Vector should be *O*(log *N*).

258/2 • the worst-case time complexity of Prepend with Count=1 and Delete_First with Count=1 should be *O*(*N* log *N*).

258.a/2 **Implementation Advice:** The worst-case time complexity of Prepend with Count = 1 and Delete_First with Count=1 for Containers.Vectors should be *O*(*N* log *N*).

258.b/2 **Reason:** We do not mean to overly constrain implementation strategies here. However, it is important for portability that the performance of large containers has roughly the same factors on different implementations. If a program is moved to an implementation that takes *O*(*N*) time to access elements, that program could be unusable when the vectors are large. We allow *O*(log *N*) access because the proportionality constant and caching effects are likely to be larger than the log factor, and we don't want to discourage innovative implementations.

259/2 {*AI95-00302-03*} The worst-case time complexity of a call on procedure Sort of an instance of Containers.Vectors.Generic_Sorting should be *O*(*N***2), and the average time complexity should be better than *O*(*N***2).

259.a/2 **Implementation Advice:** The worst-case time complexity of a call on procedure Sort of an instance of Containers.Vectors.Generic_Sorting should be *O*(*N***2), and the average time complexity should be better than *O*(*N***2).

259.b/2 **Ramification:** In other words, we're requiring the use of a better than *O*(*N***2) sorting algorithm, such as Quicksort. No bubble sorts allowed!

260/2 {*AI95-00302-03*} Containers.Vectors.Generic_Sorting.Sort and Containers.Vectors.Generic_Sorting.Merge should minimize copying of elements.

260.a/2 **Implementation Advice:** Containers.Vectors.Generic_Sorting.Sort and Containers.Vectors.Generic_Sorting.Merge should minimize copying of elements.

260.b/2 **To be honest:** We do not mean “absolutely minimize” here; we're not intending to require a single copy for each element. Rather, we want to suggest that the sorting algorithm chosen is one that does not copy items unnecessarily. Bubble sort would not meet this advice, for instance.

261/2 {*AI95-00302-03*} Move should not copy elements, and should minimize copying of internal data structures.

261.a/2 **Implementation Advice:** Containers.Vectors.Move should not copy elements, and should minimize copying of internal data structures.

261.b/2 **Implementation Note:** Usually that can be accomplished simply by moving the pointer(s) to the internal data structures from the Source vector to the Target vector.

262/2 {*AI95-00302-03*} If an exception is propagated from a vector operation, no storage should be lost, nor any elements removed from a vector unless specified by the operation.

262.a/2 **Implementation Advice:** If an exception is propagated from a vector operation, no storage should be lost, nor any elements removed from a vector unless specified by the operation.

262.b/2 **Reason:** This is important so that programs can recover from errors. But we don't want to require heroic efforts, so we just require documentation of cases where this can't be accomplished.

NOTE 1 {*AI12-0440-1*} All elements of a vector occupy locations in the internal array. If a sparse container is required, a Hashed_Map can be used rather than a vector. 263/5

NOTE 2 If Index_Type'Base'First = Index_Type'First an instance of Ada.Containers.Vectors will raise Constraint_Error. A value below Index_Type'First is required so that an empty vector has a meaningful value of Last_Index. 264/2

Discussion: This property is the main reason why only integer types (as opposed to any discrete type) are allowed as the index type of a vector. An enumeration or modular type would require a subtype in order to meet this requirement. 264.a/2

Extensions to Ada 95

{*AI95-00302-03*} The package Containers.Vectors is new. 264.b/2

Incompatibilities With Ada 2005

{*AI05-0001-1*} Subprograms Assign and Copy are added to Containers.Vectors. If an instance of Containers.Vectors is referenced in a use_clause, and an entity *E* with the same defining_identifier as a new entity in Containers.Vectors is defined in a package that is also referenced in a use_clause, the entity *E* may no longer be use-visible, resulting in errors. This should be rare and is easily fixed if it does occur. 264.c/3

Extensions to Ada 2005

{*AI05-0212-1*} Added iterator, reference, and indexing support to make vector containers more convenient to use. 264.d/3

Wording Changes from Ada 2005

{*AI05-0001-1*} Generalized the definition of Reserve_Capacity and Move. Specified which elements are read/written by stream attributes. 264.e/3

{*AI05-0022-1*} **Correction:** Added a Bounded (Run-Time) Error to cover tampering by generic actual subprograms. 264.f/3

{*AI05-0027-1*} **Correction:** Added a Bounded (Run-Time) Error to cover access to finalized vector containers. 264.g/3

{*AI05-0044-1*} **Correction:** Redefined "<" actuals to require a strict weak ordering; the old definition allowed indeterminant comparisons that would not have worked in a container. 264.h/3

{*AI05-0084-1*} **Correction:** Added a pragma Remote_Types so that containers can be used in distributed programs. 264.i/3

{*AI05-0160-1*} **Correction:** Revised the definition of invalid cursors to cover missing (and new) cases. 264.j/3

{*AI05-0265-1*} **Correction:** Defined when a container prohibits tampering in order to more clearly define where the check is made and the exception raised. 264.k/3

Inconsistencies With Ada 2012

{*AI12-0111-1*} Tampering with elements is now defined to be equivalent to tampering with cursors for ordinary containers. If a program requires tampering detection to work, it might fail in Ada 2022. Specifically, if a program requires Program_Error to be raised by a routine that (only) tampers with elements in Ada 2012 (such as Replace_Element) when called in a context that does not allow tampering with elements (such as Update_Element), the routine will work as defined instead of raising Program_Error in Ada 2022. Needless to say, this shouldn't happen outside of test programs. Note that such contexts still prohibit tampering with cursors, so routines like Insert and Delete will still raise Program_Error in this case. 264.l/5

{*AI12-0112-1*} Trying to insert or concatenate more than Count_Type'Last elements will now raise Constraint_Error rather than Capacity_Error. This is extremely unlikely to happen, as Count_Type'Last is typically at least 2**31-1, so most such vectors will exceed memory before reaching this error. 264.m/5

Incompatibilities With Ada 2012

{*AI12-0111-1*} {*AI12-0112-1*} {*AI12-0339-1*} A number of new subprograms, types, and even a nested package were added to Containers.Vectors to better support contracts and stable views. Therefore, a use clause conflict is possible; see the introduction of Annex A for more on this topic. 264.n/5

{*AI12-0005-1*} {*AI12-0212-1*} {*AI12-0400-1*} Vector objects now support aggregates. This introduces a potential incompatibility for overloaded routines, including the "&" operations defined in this package. If the Element_Type of the vector is a type that allows aggregates (such as a record type), then calls to the "&" operations with an aggregate element will become ambiguous in Ada 2022, while they would have been legal in Ada 2012. This can be fixed by qualifying the aggregate with the element type. 264.o/5

{*AI12-0400-1*} **Correction:** The Insert, Append, and Prepend operations that operate on two vectors have been renamed Insert_Vector, Append_Vector, and Prepend_Vector, respectively. This was done in order to eliminate the aggregate ambiguity for the commonly used single element Append and Insert routines. The renamed routines are rarely used in Ada 2012 code, so the impact should be minimal. 264.p/5

Extensions to Ada 2012

264.q/5 {*AI12-0196-1*} **Correction:** To_Cursor and Replace_Element are now defined such that they can be used concurrently so long as they operate on different elements. This allows some container operations to be used in parallel without separate synchronization.

264.r/5 {*AI12-0212-1*} Vectors now support indexed container aggregates, so aggregate syntax can be used to create Vectors.

264.s/5 {*AI12-0266-1*} The iterator for the entire container now can return a parallel iterator which can be used to process the container in parallel.

Wording Changes from Ada 2012

264.t/4 {*AI12-0110-1*} **Corrigendum:** Clarified that tampering checks precede all other checks made by a subprogram (but come after those associated with the call).

264.u/5 {*AI12-0112-1*} Added contracts to this container. This includes describing some of the semantics with pre- and postconditions, rather than English text. Note that the preconditions can be Suppressed (see 11.5).

264.v/5 {*AI12-0400-1*} **Correction:** Split the Append routine into two routines rather than having a single routine with a default parameter, in order that a routine with the appropriate profile for the Aggregate aspect exists. This change should not change the behavior of any existing code.

A.18.3 The Generic Package Containers.Doubly_Linked_Lists

1/2 {*AI95-00302-03*} The language-defined generic package Containers.Doubly_Linked_Lists provides private types List and Cursor, and a set of operations for each type. A list container is optimized for insertion and deletion at any position.

2/2 {*AI95-00302-03*} A doubly-linked list container object manages a linked list of internal *nodes*, each of which contains an element and pointers to the next (successor) and previous (predecessor) internal nodes. A cursor designates a particular node within a list (and by extension the element contained in that node). A cursor keeps designating the same node (and element) as long as the node is part of the container, even if the node is moved in the container.

3/2 {*AI95-00302-03*} The *length* of a list is the number of elements it contains.

Static Semantics

4/2 {*AI95-00302-03*} The generic library package Containers.Doubly_Linked_Lists has the following declaration:

5/5
```
{AI05-0084-1} {AI05-0212-1} {AI12-0112-1} with Ada.Iterator_Interfaces;
generic
   type Element_Type is private;
   with function "=" (Left, Right : Element_Type)
      return Boolean is <>;
package Ada.Containers.Doubly_Linked_Lists
   with Preelaborate, Remote_Types,
        Nonblocking, Global => in out synchronized is
```

5.a/5 **Discussion:** {*AI12-0112-1*} For discussion on the reasons and meaning of the specifications of the Global and Nonblocking aspects in this generic package, see the notes on the equivalent operations in the specification of the Containers.Vectors package (see A.18.2).

```
{AI05-0212-1} {AI12-0111-1} {AI12-0112-1} {AI12-0212-1} {AI12-0339-1} {AI12-0391-1}   6/5
{AI12-0399-1} {AI12-0400-1}    type List is tagged private
      with Constant_Indexing => Constant_Reference,
           Variable_Indexing => Reference,
           Default_Iterator  => Iterate,
           Iterator_Element  => Element_Type,
           Iterator_View     => Stable.List,
           Aggregate         => (Empty       => Empty,
                                 Add_Unnamed => Append),
           Stable_Properties => (Length,
                                 Tampering_With_Cursors_Prohibited,
                                 Tampering_With_Elements_Prohibited),
           Default_Initial_Condition =>
              Length (List) = 0 and then
              (not Tampering_With_Cursors_Prohibited (List)) and then
              (not Tampering_With_Elements_Prohibited (List)),
           Preelaborable_Initialization;

{AI12-0399-1}    type Cursor is private                                              7/5
      with Preelaborable_Initialization;

   Empty_List : constant List;                                                       8/2

   No_Element : constant Cursor;                                                     9/2

{AI05-0212-1} {AI12-0112-1}    function Has_Element (Position : Cursor) return     9.1/5
Boolean
      with Nonblocking, Global => in all, Use_Formal => null;

{AI12-0112-1}    function Has_Element (Container : List; Position : Cursor)        9.2/5
      return Boolean
      with Nonblocking, Global => null, Use_Formal => null;

{AI05-0212-1}    package List_Iterator_Interfaces is new                           9.3/3
       Ada.Iterator_Interfaces (Cursor, Has_Element);

   function "=" (Left, Right : List) return Boolean;                             10/2

{AI12-0112-1}    function Tampering_With_Cursors_Prohibited                        10.1/5
      (Container : List) return Boolean
      with Nonblocking, Global => null, Use_Formal => null;

{AI12-0112-1}    function Tampering_With_Elements_Prohibited                       10.2/5
      (Container : List) return Boolean
      with Nonblocking, Global => null, Use_Formal => null;

{AI12-0339-1}    function Empty return List                                        10.3/5
      is (Empty_List)
      with Post =>
             not Tampering_With_Elements_Prohibited (Empty'Result) and then
             not Tampering_With_Cursors_Prohibited (Empty'Result) and then
             Length (Empty'Result) = 0;

{AI12-0112-1}    function Length (Container : List) return Count_Type              11/5
      with Nonblocking, Global => null, Use_Formal => null;

{AI12-0112-1}    function Is_Empty (Container : List) return Boolean               12/5
      with Nonblocking, Global => null, Use_Formal => null,
           Post => Is_Empty'Result = (Length (Container) = 0);

{AI12-0112-1}    procedure Clear (Container : in out List)                         13/5
      with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                     or else raise Program_Error,
           Post => Length (Container) = 0;

{AI12-0112-1}    function Element (Position : Cursor) return Element_Type          14/5
      with Pre  => Position /= No_Element or else raise Constraint_Error,
           Nonblocking, Global => in all, Use_Formal => Element_Type;
```

14.1/5
```
{AI12-0112-1}    function Element (Container : List;
                                   Position  : Cursor) return Element_Type
      with Pre => (Position /= No_Element or else
                      raise Constraint_Error) and then
                   (Has_Element (Container, Position)
                      or else raise Program_Error),
           Nonblocking, Global => null, Use_Formal => Element_Type;
```

15/5
```
{AI12-0112-1}    procedure Replace_Element (Container : in out List;
                                            Position  : in     Cursor;
                                            New_item  : in     Element_Type)
      with Pre  => (not Tampering_With_Elements_Prohibited (Container)
                      or else raise Program_Error) and then
                   (Position /= No_Element
                      or else raise Constraint_Error) and then
                   (Has_Element (Container, Position)
                      or else raise Program_Error);
```

16/5
```
{AI12-0112-1}    procedure Query_Element
     (Position : in Cursor;
      Process  : not null access procedure (Element : in Element_Type))
      with Pre  => Position /= No_Element or else raise Constraint_Error,
           Global => in all;
```

16.1/5
```
{AI12-0112-1}    procedure Query_Element
     (Container : in List;
      Position  : in Cursor;
      Process   : not null access procedure (Element : in Element_Type))
      with Pre  => (Position /= No_Element
                       or else raise Constraint_Error) and then
                   (Has_Element (Container, Position)
                      or else raise Program_Error);
```

17/5
```
{AI12-0112-1}    procedure Update_Element
     (Container : in out List;
      Position  : in     Cursor;
      Process   : not null access procedure
                       (Element : in out Element_Type))
      with Pre  => (Position /= No_Element
                       or else raise Constraint_Error) and then
                   (Has_Element (Container, Position)
                      or else raise Program_Error);
```

17.1/5
```
{AI05-0212-1} {AI12-0112-1}    type Constant_Reference_Type
        (Element : not null access constant Element_Type) is private
      with Implicit_Dereference => Element,
           Nonblocking, Global => in out synchronized,
           Default_Initial_Condition => (raise Program_Error);
```

17.2/5
```
{AI05-0212-1} {AI12-0112-1}    type Reference_Type (Element : not null access
Element_Type) is private
      with Implicit_Dereference => Element,
           Nonblocking, Global => in out synchronized,
           Default_Initial_Condition => (raise Program_Error);
```

17.3/5
```
{AI05-0212-1} {AI12-0112-1}    function Constant_Reference (Container : aliased in
List;
                                Position  : in Cursor)
      return Constant_Reference_Type
      with Pre  => (Position /= No_Element or else
                       raise Constraint_Error) and then
                   (Has_Element (Container, Position) or else
                       raise Program_Error),
           Post   => Tampering_With_Cursors_Prohibited (Container),
           Nonblocking, Global => null, Use_Formal => null;
```

```
{AI05-0212-1} {AI12-0112-1}    function Reference (Container : aliased in out List;                17.4/5
                      Position  : in Cursor)
      return Reference_Type
      with Pre  => (Position /= No_Element or else
                      raise Constraint_Error) and then
                   (Has_Element (Container, Position) or else
                      raise Program_Error),
           Post   => Tampering_With_Cursors_Prohibited (Container),
           Nonblocking, Global => null, Use_Formal => null;

{AI05-0001-1} {AI12-0112-1}    procedure Assign (Target : in out List; Source : in                 17.5/5
List)
      with Pre  => not Tampering_With_Cursors_Prohibited (Target)
                     or else raise Program_Error,
           Post => Length (Source) = Length (Target);

{AI05-0001-1} {AI12-0112-1}    function Copy (Source : List)                                       17.6/5
      return List
      with Post =>
             Length (Copy'Result) = Length (Source) and then
             not Tampering_With_Elements_Prohibited (Copy'Result) and then
             not Tampering_With_Cursors_Prohibited (Copy'Result);

{AI12-0112-1}    procedure Move (Target : in out List;                                             18/5
                   Source : in out List)
      with Pre  => (not Tampering_With_Cursors_Prohibited (Target)
                      or else raise Program_Error) and then
                   (not Tampering_With_Cursors_Prohibited (Source)
                      or else raise Program_Error),
            Post => (if not Target'Has_Same_Storage (Source) then
                       Length (Target) = Length (Source'Old) and then
                       Length (Source) = 0);

{AI12-0112-1}    procedure Insert (Container : in out List;                                        19/5
                     Before    : in     Cursor;
                     New_Item  : in     Element_Type;
                     Count     : in     Count_Type := 1)
      with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                      or else raise Program_Error) and then
                   (Before = No_Element or else
                    Has_Element (Container, Before)
                      or else raise Program_Error) and then
                   (Length (Container) <= Count_Type'Last - Count
                      or else raise Constraint_Error),
           Post => Length (Container)'Old + Count = Length (Container);

{AI12-0112-1}    procedure Insert (Container : in out List;                                        20/5
                     Before    : in     Cursor;
                     New_Item  : in     Element_Type;
                     Position  :    out Cursor;
                     Count     : in     Count_Type := 1)
      with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                      or else raise Program_Error) and then
                   (Before = No_Element or else
                    Has_Element (Container, Before)
                      or else raise Program_Error) and then
                   (Length (Container) <= Count_Type'Last - Count
                      or else raise Constraint_Error),
           Post => Length (Container)'Old + Count = Length (Container)
                   and then Has_Element (Container, Position);
```

21/5 {*AI12-0112-1*} **procedure** Insert (Container : **in out** List;
Before : **in** Cursor;
Position : **out** Cursor;
Count : **in** Count_Type := 1)
with Pre => (**not** Tampering_With_Cursors_Prohibited (Container)
or else raise Program_Error) **and then**
(Before = No_Element **or else**
Has_Element (Container, Before)
or else raise Program_Error) **and then**
(Length (Container) <= Count_Type'Last - Count
or else raise Constraint_Error),
Post => Length (Container)'Old + Count = Length (Container)
and then Has_Element (Container, Position);

22/5 {*AI12-0112-1*} **procedure** Prepend (Container : **in out** List;
New_Item : **in** Element_Type;
Count : **in** Count_Type := 1)
with Pre => (**not** Tampering_With_Cursors_Prohibited (Container)
or else raise Program_Error) **and then**
(Length (Container) <= Count_Type'Last - Count
or else raise Constraint_Error),
Post => Length (Container)'Old + Count = Length (Container);

23/5 {*AI12-0112-1*} {*AI12-0400-1*} **procedure** Append (Container : **in out** List;
New_Item : **in** Element_Type;
Count : **in** Count_Type)
with Pre => (**not** Tampering_With_Cursors_Prohibited (Container)
or else raise Program_Error) **and then**
(Length (Container) <= Count_Type'Last - Count
or else raise Constraint_Error),
Post => Length (Container)'Old + Count = Length (Container);

23.1/5 {*AI12-0391-1*} {*AI12-0400-1*} **procedure** Append (Container : **in out** List;
New_Item : **in** Element_Type)
with Pre => (**not** Tampering_With_Cursors_Prohibited (Container)
or else raise Program_Error) **and then**
(Length (Container) <= Count_Type'Last - 1
or else raise Constraint_Error),
Post => Length (Container)'Old + 1 = Length (Container);

24/5 {*AI12-0112-1*} **procedure** Delete (Container : **in out** List;
Position : **in out** Cursor;
Count : **in** Count_Type := 1)
with Pre => (**not** Tampering_With_Cursors_Prohibited (Container)
or else raise Program_Error) **and then**
(Position /= No_Element
or else raise Constraint_Error) **and then**
(Has_Element (Container, Position)
or else raise Program_Error),
Post => Length (Container)'Old - Count <= Length (Container)
and then Position = No_Element;

25/5 {*AI12-0112-1*} **procedure** Delete_First (Container : **in out** List;
Count : **in** Count_Type := 1)
with Pre => **not** Tampering_With_Cursors_Prohibited (Container)
or else raise Program_Error,
Post => Length (Container)'Old - Count <= Length (Container);

26/5 {*AI12-0112-1*} **procedure** Delete_Last (Container : **in out** List;
Count : **in** Count_Type := 1)
with Pre => **not** Tampering_With_Cursors_Prohibited (Container)
or else raise Program_Error,
Post => Length (Container)'Old - Count <= Length (Container);

27/5 {*AI12-0112-1*} **procedure** Reverse_Elements (Container : **in out** List)
with Pre => **not** Tampering_With_Cursors_Prohibited (Container)
or else raise Program_Error;

```
{AI12-0112-1}    procedure Swap (Container : in out List;                                    28/5
                   I, J      : in     Cursor)
      with Pre  => (not Tampering_With_Elements_Prohibited (Container)
                      or else raise Program_Error) and then
                   (I /= No_Element or else Constraint_Error) and then
                   (J /= No_Element or else Constraint_Error) and then
                   (Has_Element (Container, I)
                      or else raise Program_Error) and then
                   (Has_Element (Container, J)
                      or else raise Program_Error);
{AI12-0112-1}    procedure Swap_Links (Container : in out List;                              29/5
                          I, J      : in     Cursor)
      with Pre  => (not Tampering_With_Elements_Prohibited (Container)
                       or else raise Program_Error) and then
                   (I /= No_Element or else Constraint_Error) and then
                   (J /= No_Element or else Constraint_Error) and then
                   (Has_Element (Container, I)
                       or else raise Program_Error) and then
                   (Has_Element (Container, J)
                       or else raise Program_Error);
{AI12-0112-1}    procedure Splice (Target   : in out List;                                   30/5
                       Before   : in     Cursor;
                       Source   : in out List)
      with Pre  => (not Tampering_With_Cursors_Prohibited (Target)
                      or else raise Program_Error) and then
                   (not Tampering_With_Cursors_Prohibited (Source)
                      or else raise Program_Error) and then
                   (Before = No_Element or else
                    Has_Element (Target, Before)
                      or else raise Program_Error) and then
                   (Target'Has_Same_Storage (Source) or else
                    Length (Target) <= Count_Type'Last - Length (Source)
                      or else raise Constraint_Error),
           Post => (if not Target'Has_Same_Storage (Source) then
                      (declare
                         Result_Length : constant Count_Type :=
                            Length (Source)'Old + Length (Target)'Old;
                       begin
                         Length (Source) = 0 and then
                         Length (Target) = Result_Length));
```

31/5 {*AI12-0112-1*} **procedure** Splice (Target : **in out** List;
Before : **in** Cursor;
Source : **in out** List;
Position : **in out** Cursor)
with Pre => (**not** Tampering_With_Cursors_Prohibited (Target)
or else raise Program_Error) **and then**
(**not** Tampering_With_Cursors_Prohibited (Source)
or else raise Program_Error) **and then**
(Position /= No_Element
or else raise Constraint_Error) **and then**
(Has_Element (Source, Position)
or else raise Program_Error) **and then**
(Before = No_Element **or else**
Has_Element (Target, Before)
or else raise Program_Error) **and then**
(Target'Has_Same_Storage (Source) **or else**
Length (Target) <= Count_Type'Last - 1
or else raise Constraint_Error),
Post => (**declare**
Org_Target_Length : **constant** Count_Type :=
Length (Target)'Old;
Org_Source_Length : **constant** Count_Type :=
Length (Source)'Old;
begin
(**if** Target'Has_Same_Storage (Source) **then**
Position = Position'Old
else
Length (Source) = Org_Source_Length - 1 **and then**
Length (Target) = Org_Target_Length + 1 **and then**
Has_Element (Target, Position)));

32/2 **procedure** Splice (Container: **in out** List;
Before : **in** Cursor;
Position : **in** Cursor)
with Pre => (**not** Tampering_With_Cursors_Prohibited (Container)
or else raise Program_Error) **and then**
(Position /= No_Element
or else raise Constraint_Error) **and then**
(Has_Element (Container, Position)
or else raise Program_Error) **and then**
(Before = No_Element **or else**
Has_Element (Container, Before)
or else raise Program_Error),
Post => Length (Container) = Length (Container)'Old;

33/5 {*AI12-0112-1*} **function** First (Container : List) **return** Cursor
with Nonblocking, Global => **null**, Use_Formal => **null**,
Post => (**if not** Is_Empty (Container)
then Has_Element (Container, First'Result)
else First'Result = No_Element);

34/5 {*AI12-0112-1*} **function** First_Element (Container : List)
return Element_Type
with Pre => (**not** Is_Empty (Container)
or else raise Constraint_Error);

35/5 {*AI12-0112-1*} **function** Last (Container : List) **return** Cursor
with Nonblocking, Global => **null**, Use_Formal => **null**,
Post => (**if** not Is_Empty (Container)
then Has_Element (Container, Last'Result)
else Last'Result = No_Element);

36/5 {*AI12-0112-1*} **function** Last_Element (Container : List)
return Element_Type
with Pre => (**not** Is_Empty (Container)
or else raise Constraint_Error);

37/5 {*AI12-0112-1*} **function** Next (Position : Cursor) **return** Cursor
with Nonblocking, Global => **in all**, Use_Formal => **null**,
Post => (**if** Position = No_Element **then** Next'Result = No_Element);

```
{AI12-0112-1}    function Next (Container : List;                                    37.1/5
                  Position : Cursor) return Cursor
      with Nonblocking, Global => null, Use_Formal => null,
           Pre  => Position = No_Element or else
                   Has_Element (Container, Position)
                      or else raise Program_Error,
           Post => (if Position = No_Element then Next'Result = No_Element
                    elsif Next'Result = No_Element then
                      Position = Last (Container)
                    else Has_Element (Container, Next'Result));

{AI12-0112-1}    function Previous (Position : Cursor) return Cursor                 38/5
      with Nonblocking, Global => in all, Use_Formal => null,
           Post => (if Position = No_Element then
                      Previous'Result = No_Element);

{AI12-0112-1}    function Previous (Container : List;                                38.1/5
                        Position  : Cursor) return Cursor
      with Nonblocking, Global => null, Use_Formal => null,
           Pre  => Position = No_Element or else
                   Has_Element (Container, Position)
                      or else raise Program_Error,
           Post => (if Position = No_Element then
                      Previous'Result = No_Element
                    elsif Previous'Result = No_Element then
                      Position = First (Container)
                    else Has_Element (Container, Previous'Result));

{AI12-0112-1}    procedure Next (Position : in out Cursor)                           39/5
      with Nonblocking, Global => in all, Use_Formal => null;

{AI12-0112-1}    procedure Next (Container : in     List;                           39.1/5
                   Position  : in out Cursor)
      with Nonblocking, Global => null, Use_Formal => null,
           Pre  => Position = No_Element or else
                   Has_Element (Container, Position)
                      or else raise Program_Error,
           Post => (if Position /= No_Element
                    then Has_Element (Container, Position));

{AI12-0112-1}    procedure Previous (Position : in out Cursor)                       40/5
      with Nonblocking, Global => in all, Use_Formal => null;

{AI12-0112-1}    procedure Previous (Container : in     List;                       40.1/5
                     Position  : in out Cursor)
      with Nonblocking, Global => null, Use_Formal => null,
           Pre  => Position = No_Element or else
                   Has_Element (Container, Position)
                      or else raise Program_Error,
           Post => (if Position /= No_Element then
                      Has_Element (Container, Position));

{AI12-0112-1}    function Find (Container : List;                                    41/5
                   Item      : Element_Type;
                   Position  : Cursor := No_Element)
      return Cursor
      with Pre  => Position = No_Element or else
                   Has_Element (Container, Position)
                      or else raise Program_Error,
           Post => (if Find'Result /= No_Element
                    then Has_Element (Container, Find'Result));

{AI12-0112-1}    function Reverse_Find (Container : List;                            42/5
                          Item      : Element_Type;
                          Position  : Cursor := No_Element)
      return Cursor
      with Pre  => Position = No_Element or else
                   Has_Element (Container, Position)
                      or else raise Program_Error,
           Post => (if Reverse_Find'Result /= No_Element
                    then Has_Element (Container, Reverse_Find'Result));
```

43/2
```
   function Contains (Container : List;
                      Item      : Element_Type) return Boolean;
```

44/3 *This paragraph was deleted.*{*AI05-0212-1*}

45/5
```
{AI12-0112-1}    procedure  Iterate
     (Container : in List;
      Process   : not null access procedure (Position : in Cursor))
      with Allows_Exit;
```

46/5
```
{AI12-0112-1}    procedure Reverse_Iterate
     (Container : in List;
      Process   : not null access procedure (Position : in Cursor))
      with Allows_Exit;
```

46.1/5
```
{AI05-0212-1} {AI12-0112-1} {AI12-0266-1}    function Iterate (Container : in List)
      return List_Iterator_Interfaces.Parallel_Reversible_Iterator'Class
      with Post   => Tampering_With_Cursors_Prohibited (Container);
```

46.2/5
```
{AI05-0212-1} {AI12-0112-1}    function Iterate (Container : in List; Start : in
Cursor)
      return List_Iterator_Interfaces.Reversible_Iterator'Class
      with Pre    => (Start /= No_Element
                        or else raise Constraint_Error) and then
                     (Has_Element (Container, Start)
                        or else raise Program_Error),
           Post   => Tampering_With_Cursors_Prohibited (Container);
```

47/5
```
{AI12-0112-1}    generic
      with function "<" (Left, Right : Element_Type)
         return Boolean is <>;
   package Generic_Sorting
   with Nonblocking, Global => null is
```

48/2
```
      function Is_Sorted (Container : List) return Boolean;
```

49/5
```
{AI12-0112-1}       procedure Sort (Container : in out List)
         with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                        or else raise Program_Error;
```

50/5
```
{AI12-0112-1}       procedure Merge (Target  : in out List;
                                    Source  : in out List)
         with Pre  => (not Tampering_With_Cursors_Prohibited (Target)
                        or else raise Program_Error) and then
                      (not Tampering_With_Elements_Prohibited (Source)
                        or else raise Program_Error) and then
                      (Length (Target) <= Count_Type'Last - Length (Source)
                        or else raise Constraint_Error) and then
                      ((Length (Source) = 0 or else
                        not Target'Has_Same_Storage (Source))
                        or else raise Constraint_Error),
              Post => (declare
                         Result_Length : constant Count_Type :=
                            Length (Source)'Old + Length (Target)'Old;
                       begin
                         (Length (Source) = 0 and then
                          Length (Target) = Result_Length));
```

51/2
```
   end Generic_Sorting;
```

51.1/5
```
{AI12-0111-1}    package Stable is
```

51.2/5
```
{AI12-0111-1} {AI12-0339-1} {AI12-0391-1} {AI12-0400-1} {AI12-0407-1}       type
List (Base : not null access Doubly_Linked_Lists.List) is
         tagged limited private
         with Constant_Indexing => Constant_Reference,
              Variable_Indexing => Reference,
              Default_Iterator  => Iterate,
              Iterator_Element  => Element_Type,
              Stable_Properties => (Length),
              Global => null,
              Default_Initial_Condition => Length (List) = 0,
              Preelaborable_Initialization;
```

```
{AI12-0111-1}       type Cursor is private                                          51.3/5
         with Preelaborable_Initialization;
{AI12-0111-1}       Empty_List : constant List;                                     51.4/5
{AI12-0111-1}       No_Element : constant Cursor;                                   51.5/5
{AI12-0111-1}       function Has_Element (Position : Cursor) return Boolean         51.6/5
         with Nonblocking, Global => in all, Use_Formal => null;
{AI12-0111-1}       package List_Iterator_Interfaces is new                         51.7/5
         Ada.Iterator_Interfaces (Cursor, Has_Element);
{AI12-0111-1}       procedure Assign (Target : in out Doubly_Linked_Lists.List;     51.8/5
                         Source : in List)
         with Post => Length (Source) = Length (Target);
{AI12-0111-1}       function Copy (Source : Doubly_Linked_Lists.List) return List   51.9/5
         with Post => Length (Copy'Result) = Length (Source);
{AI12-0111-1}       type Constant_Reference_Type                                    51.10/5
            (Element : not null access constant Element_Type) is private
         with Implicit_Dereference => Element,
              Nonblocking, Global => null, Use_Formal => null,
              Default_Initial_Condition => (raise Program_Error);
{AI12-0111-1}       type Reference_Type                                             51.11/5
            (Element : not null access Element_Type) is private
         with Implicit_Dereference => Element,
              Nonblocking, Global => null, Use_Formal => null,
              Default_Initial_Condition => (raise Program_Error);
{AI12-0111-1}       -- Additional subprograms as described in the text              51.12/5
      -- are declared here.
{AI12-0111-1}    private                                                            51.13/5
{AI12-0111-1}       ... -- not specified by the language                            51.14/5
{AI12-0111-1}    end Stable;                                                        51.15/5
private                                                                            52/2
   ... -- not specified by the language                                            53/2
end Ada.Containers.Doubly_Linked_Lists;                                            54/2
```

{*AI95-00302-03*} The actual function for the generic formal function "=" on Element_Type values is expected to define a reflexive and symmetric relationship and return the same result value each time it is called with a particular pair of values. If it behaves in some other manner, the functions Find, Reverse_Find, and "=" on list values return an unspecified value. The exact arguments and number of calls of this generic formal function by the functions Find, Reverse_Find, and "=" on list values are unspecified. 55/2

Ramification: If the actual function for "=" is not symmetric and consistent, the result returned by the listed functions cannot be predicted. The implementation is not required to protect against "=" raising an exception, or returning random results, or any other “bad” behavior. And it can call "=" in whatever manner makes sense. But note that only the results of Find, Reverse_Find, and List "=" are unspecified; other subprograms are not allowed to break if "=" is bad (they aren't expected to use "="). 55.a/2

{*AI95-00302-03*} The type List is used to represent lists. The type List needs finalization (see 7.6). 56/2

{*AI95-00302-03*} Empty_List represents the empty List object. It has a length of 0. If an object of type List is not otherwise initialized, it is initialized to the same value as Empty_List. 57/2

{*AI95-00302-03*} No_Element represents a cursor that designates no element. If an object of type Cursor is not otherwise initialized, it is initialized to the same value as No_Element. 58/2

{*AI95-00302-03*} {*AI12-0434-1*} The primitive "=" operator for type Cursor returns True if both cursors are No_Element, or designate the same element in the same container. 59/5

To be honest: {*AI12-0434-1*} “The primitive "=" operator” is the one with two parameters of type Cursor which returns Boolean. We're not talking about some other (hidden) primitive function named "=". 59.a/5

60/2 {*AI95-00302-03*} Execution of the default implementation of the Input, Output, Read, or Write attribute of type Cursor raises Program_Error.

60.a/2 **Reason:** A cursor will probably be implemented in terms of one or more access values, and the effects of streaming access values is unspecified. Rather than letting the user stream junk by accident, we mandate that streaming of cursors raise Program_Error by default. The attributes can always be specified if there is a need to support streaming.

60.1/5 {*AI05-0001-1*} {*AI05-0262-1*} {*AI12-0437-1*} List'Write for a List object *L* writes Length(*L*) elements of the list to the stream. It may also write additional information about the list.

60.2/3 {*AI05-0001-1*} {*AI05-0262-1*} List'Read reads the representation of a list from the stream, and assigns to *Item* a list with the same length and elements as was written by List'Write.

60.b/3 **Ramification:** Streaming more elements than the container length is wrong. For implementation implications of this rule, see the Implementation Note in A.18.2.

61/5 {*AI95-00302-03*} {*AI12-0111-1*} {*AI12-0112-1*} [Some operations check for “tampering with cursors” of a container because they depend on the set of elements of the container remaining constant, and others check for “tampering with elements” of a container because they depend on elements of the container not being replaced.] When tampering with cursors is *prohibited* for a particular list object *L*, Program_Error is propagated by the finalization of *L*[, as well as by a call that passes *L* to certain of the operations of this package, as indicated by the precondition of such an operation]. Similarly, when tampering with elements is *prohibited* for *L*, Program_Error is propagated by a call that passes *L* to certain of the other operations of this package, as indicated by the precondition of such an operation.

Paragraphs 62 through 69 are removed as preconditions now describe these rules.

65.a.1/3 **Ramification:** We don't need to explicitly mention assignment_statement, because that finalizes the target object as part of the operation, and finalization of an object is already defined as tampering with cursors.

69.2/5
```
function Has_Element (Position : Cursor) return Boolean
   with Nonblocking, Global => in all, Use_Formal => null;
```

69.3/3 {*AI05-0212-1*} Returns True if Position designates an element, and returns False otherwise.

69.c/3 **To be honest:** {*AI05-0005-1*} {*AI05-0212-1*} This function might not detect cursors that designate deleted elements; such cursors are invalid (see below) and the result of calling Has_Element with an invalid cursor is unspecified (but not erroneous).

69.4/5
```
function Has_Element (Container : List; Position : Cursor)
   return Boolean
   with Nonblocking, Global => null, Use_Formal => null;
```

69.5/5 {*AI12-0112-1*} Returns True if Position designates an element in Container, and returns False otherwise.

69.d/5 **Ramification:** If Position is No_Element, Has_Element returns False.

70/2
```
function "=" (Left, Right : List) return Boolean;
```

71/3 {*AI95-00302-03*} {*AI05-0264-1*} If Left and Right denote the same list object, then the function returns True. If Left and Right have different lengths, then the function returns False. Otherwise, it compares each element in Left to the corresponding element in Right using the generic formal equality operator. If any such comparison returns False, the function returns False; otherwise, it returns True. Any exception raised during evaluation of element equality is propagated.

71.a/2 **Implementation Note:** This wording describes the canonical semantics. However, the order and number of calls on the formal equality function is unspecified for all of the operations that use it in this package, so an implementation can call it as many or as few times as it needs to get the correct answer. Specifically, there is no requirement to call the formal equality additional times once the answer has been determined.

```
function Tampering_With_Cursors_Prohibited                                    71.1/5
   (Container : List) return Boolean
   with Nonblocking, Global => null, Use_Formal => null;
```

{*AI12-0112-1*} Returns True if tampering with cursors or tampering with elements is currently prohibited for Container, and returns False otherwise. 71.2/5

Reason: {*AI12-0112-1*} Prohibiting tampering with elements also needs to prohibit tampering with cursors, as deleting an element is similar to replacing it. 71.b/5

Implementation Note: {*AI12-0112-1*} Various contracts elsewhere in this specification require that this function be implemented with synchronized data. Moreover, it is possible for tampering to be prohibited by multiple operations (sequentially or in parallel). Therefore, tampering needs to be implemented with an atomic or protected counter. The counter is initialized to zero, and is incremented when tampering is prohibited, and decremented when leaving an area that prohibited tampering. Function Tampering_With_Cursors_Prohibited returns True if the counter is nonzero. (Note that any case where the result is not well-defined for one task is incorrect use of shared variables and would be erroneous by the rules of 9.10, so no special protection is needed to read the counter.) 71.c/5

```
function Tampering_With_Elements_Prohibited                                   71.3/5
   (Container : List) return Boolean
   with Nonblocking, Global => null, Use_Formal => null;
```

{*AI12-0112-1*} Always returns False[, regardless of whether tampering with elements is prohibited]. 71.4/5

Reason: {*AI12-0111-1*} A definite element cannot change size, so we allow operations that tamper with elements even when tampering with elements is prohibited. That's not true for the indefinite containers, which is why this kind of tampering exists. 71.d/5

```
function Length (Container : List) return Count_Type                          72/5
   with Nonblocking, Global => null, Use_Formal => null;
```

{*AI95-00302-03*} Returns the number of elements in Container. 73/2

```
function Is_Empty (Container : List) return Boolean                           74/5
   with Nonblocking, Global => null, Use_Formal => null,
        Post => Is_Empty'Result = (Length (Container) = 0);
```

{*AI95-00302-03*} {*AI12-0112-1*} Returns True if Container is empty. 75/5

```
procedure Clear (Container : in out List)                                     76/5
   with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error,
        Post => Length (Container) = 0;
```

{*AI95-00302-03*} Removes all the elements from Container. 77/2

```
function Element (Position : Cursor) return Element_Type                      78/5
   with Pre  => Position /= No_Element or else raise Constraint_Error,
        Nonblocking, Global => in all, Use_Formal => Element_Type;
```

{*AI95-00302-03*} {*AI12-0112-1*} Element returns the element designated by Position. 79/5

```
function Element (Container : List;                                           79.1/5
                  Position  : Cursor) return Element_Type
   with Pre => (Position /= No_Element or else
                   raise Constraint_Error) and then
               (Has_Element (Container, Position)
                   or else raise Program_Error),
        Nonblocking, Global => null, Use_Formal => Element_Type;
```

{*AI12-0112-1*} Element returns the element designated by Position in Container. 79.2/5

80/5
```
procedure Replace_Element (Container : in out List;
                           Position  : in     Cursor;
                           New_item  : in     Element_Type)
   with Pre  => (not Tampering_With_Elements_Prohibited (Container)
                    or else raise Program_Error) and then
                 (Position /= No_Element
                    or else raise Constraint_Error) and then
                 (Has_Element (Container, Position)
                    or else raise Program_Error);
```

81/5 {*AI95-00302-03*} {*AI05-0264-1*} {*AI12-0112-1*} {*AI12-0196-1*} Replace_Element assigns the value New_Item to the element designated by Position. For the purposes of determining whether the parameters overlap in a call to Replace_Element, the Container parameter is not considered to overlap with any object [(including itself)].

82/5
```
procedure Query_Element
  (Position : in Cursor;
   Process  : not null access procedure (Element : in Element_Type))
   with Pre  => Position /= No_Element or else raise Constraint_Error,
        Global => in all;
```

83/5 {*AI95-00302-03*} {*AI05-0021-1*} {*AI05-0265-1*} {*AI12-0112-1*} Query_Element calls Process.**all** with the element designated by Position as the argument. Tampering with the elements of the list that contains the element designated by Position is prohibited during the execution of the call on Process.**all**. Any exception raised by Process.**all** is propagated.

83.1/5
```
procedure Query_Element
  (Container : in List;
   Position  : in Cursor;
   Process   : not null access procedure (Element : in Element_Type))
   with Pre  => (Position /= No_Element
                    or else raise Constraint_Error) and then
                (Has_Element (Container, Position)
                    or else raise Program_Error);
```

83.2/5 {*AI12-0112-1*} Query_Element calls Process.**all** with the element designated by Position as the argument. Tampering with the elements of Container is prohibited during the execution of the call on Process.**all**. Any exception raised by Process.**all** is propagated.

84/5
```
procedure Update_Element
  (Container : in out List;
   Position  : in     Cursor;
   Process   : not null access procedure
                  (Element : in out Element_Type))
   with Pre  => (Position /= No_Element
                    or else raise Constraint_Error) and then
                (Has_Element (Container, Position)
                    or else raise Program_Error);
```

85/5 {*AI95-00302-03*} {*AI05-0264-1*} {*AI05-0265-1*} {*AI12-0112-1*} Update_Element calls Process.**all** with the element designated by Position as the argument. Tampering with the elements of Container is prohibited during the execution of the call on Process.**all**. Any exception raised by Process.**all** is propagated.

86/2 If Element_Type is unconstrained and definite, then the actual Element parameter of Process.**all** shall be unconstrained.

86.a/2 **Ramification:** This means that the elements cannot be directly allocated from the heap; it must be possible to change the discriminants of the element in place.

```
{AI12-0112-1} type Constant_Reference_Type                                    86.1/5
      (Element : not null access constant Element_Type) is private
   with Implicit_Dereference => Element,
        Nonblocking, Global => in out synchronized,
        Default_Initial_Condition => (raise Program_Error);

{AI12-0112-1} type Reference_Type (Element : not null access Element_Type) is  86.2/5
private
   with Implicit_Dereference => Element,
        Nonblocking, Global => in out synchronized,
        Default_Initial_Condition => (raise Program_Error);
```

{*AI05-0212-1*} The types Constant_Reference_Type and Reference_Type need finalization. 86.3/3

This paragraph was deleted.{*AI12-0112-1*} 86.4/5

Reason: It is expected that Reference_Type (and Constant_Reference_Type) will be a controlled type, for which finalization will have some action to terminate the tampering check for the associated container. If the object is created by default, however, there is no associated container. Since this is useless, and supporting this case would take extra work, we define it to raise an exception. 86.b/3

```
{AI12-0112-1} function Constant_Reference (Container : aliased in List;      86.5/5
                           Position  : in Cursor)
   return Constant_Reference_Type
   with Pre  => (Position /= No_Element or else
                    raise Constraint_Error) and then
                 (Has_Element (Container, Position) or else
                    raise Program_Error),
        Post   => Tampering_With_Cursors_Prohibited (Container),
        Nonblocking, Global => null, Use_Formal => null;
```

{*AI05-0212-1*} {*AI05-0269-1*} This function (combined with the Constant_Indexing and Implicit_Dereference aspects) provides a convenient way to gain read access to an individual element of a list given a cursor. 86.6/3

{*AI05-0212-1*} {*AI05-0265-1*} {*AI12-0112-1*} Constant_Reference returns an object whose discriminant is an access value that designates the element designated by Position. Tampering with the elements of Container is prohibited while the object returned by Constant_Reference exists and has not been finalized. 86.7/5

```
{AI12-0112-1} function Reference (Container : aliased in out List;           86.8/5
                     Position  : in Cursor)
   return Reference_Type
   with Pre  => (Position /= No_Element or else
                    raise Constraint_Error) and then
                 (Has_Element (Container, Position) or else
                    raise Program_Error),
        Post   => Tampering_With_Cursors_Prohibited (Container),
        Nonblocking, Global => null, Use_Formal => null;
```

{*AI05-0212-1*} {*AI05-0269-1*} This function (combined with the Variable_Indexing and Implicit_Dereference aspects) provides a convenient way to gain read and write access to an individual element of a list given a cursor. 86.9/3

{*AI05-0212-1*} {*AI05-0265-1*} {*AI12-0112-1*} Reference returns an object whose discriminant is an access value that designates the element designated by Position. Tampering with the elements of Container is prohibited while the object returned by Reference exists and has not been finalized. 86.10/5

86.11/5
```
{AI12-0112-1} procedure Assign (Target : in out List; Source : in List)
   with Pre  => not Tampering_With_Cursors_Prohibited (Target)
                    or else raise Program_Error,
        Post => Length (Source) = Length (Target);
```

86.12/3 {*AI05-0001-1*} {*AI05-0248-1*} If Target denotes the same object as Source, the operation has no effect. Otherwise, the elements of Source are copied to Target as for an assignment_statement assigning Source to Target.

86.c/3 **Discussion:** {*AI05-0005-1*} This routine exists for compatibility with the bounded list container. For an unbounded list, `Assign(A, B)` and `A := B` behave identically. For a bounded list, := will raise an exception if the container capacities are different, while Assign will not raise an exception if there is enough room in the target.

86.13/5
```
{AI12-0112-1} function Copy (Source : List)
   return List
   with Post =>
           Length (Copy'Result) = Length (Source) and then
           not Tampering_With_Elements_Prohibited (Copy'Result) and then
           not Tampering_With_Cursors_Prohibited (Copy'Result);
```

86.14/3 {*AI05-0001-1*} Returns a list whose elements match the elements of Source.

87/5
```
{AI12-0112-1} procedure Move (Target : in out List;
                              Source : in out List)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Target)
                     or else raise Program_Error) and then
                (not Tampering_With_Cursors_Prohibited (Source)
                     or else raise Program_Error),
        Post => (if not Target'Has_Same_Storage (Source) then
                     Length (Target) = Length (Source'Old) and then
                     Length (Source) = 0);
```

88/3 {*AI95-00302-03*} {*AI05-0001-1*} {*AI05-0248-1*} {*AI05-0262-1*} If Target denotes the same object as Source, then the operation has no effect. Otherwise, the operation is equivalent to Assign (Target, Source) followed by Clear (Source).

89/5
```
procedure Insert (Container : in out List;
                  Before    : in     Cursor;
                  New_Item  : in     Element_Type;
                  Count     : in     Count_Type := 1)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                     or else raise Program_Error) and then
                (Before = No_Element or else
                 Has_Element (Container, Before)
                     or else raise Program_Error) and then
                (Length (Container) <= Count_Type'Last - Count
                     or else raise Constraint_Error),
        Post => Length (Container)'Old + Count = Length (Container);
```

90/5 {*AI95-00302-03*} {*AI12-0112-1*} Insert inserts Count copies of New_Item prior to the element designated by Before. If Before equals No_Element, the new elements are inserted after the last node (if any). Any exception raised during allocation of internal storage is propagated, and Container is not modified.

90.a/2 **Ramification:** The check on Before checks that the cursor does not belong to some other Container. This check implies that a reference to the container is included in the cursor value. This wording is not meant to require detection of dangling cursors; such cursors are defined to be invalid, which means that execution is erroneous, and any result is allowed (including not raising an exception).

```
procedure Insert (Container : in out List;                                    91/5
                  Before    : in     Cursor;
                  New_Item  : in     Element_Type;
                  Position  :    out Cursor;
                  Count     : in     Count_Type := 1)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error) and then
                 (Before = No_Element or else
                  Has_Element (Container, Before)
                    or else raise Program_Error) and then
                 (Length (Container) <= Count_Type'Last - Count
                    or else raise Constraint_Error),
        Post => Length (Container)'Old + Count = Length (Container)
                and then Has_Element (Container, Position);
```

{*AI95-00302-03*} {*AI05-0257-1*} {*AI12-0112-1*} Insert allocates Count copies of New_Item, and inserts them prior to the element designated by Before. If Before equals No_Element, the new elements are inserted after the last element (if any). Position designates the first newly-inserted element, or if Count equals 0, then Position is assigned the value of Before. Any exception raised during allocation of internal storage is propagated, and Container is not modified. 92/5

```
procedure Insert (Container : in out List;                                    93/5
                  Before    : in     Cursor;
                  Position  :    out Cursor;
                  Count     : in     Count_Type := 1)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error) and then
                 (Before = No_Element or else
                  Has_Element (Container, Before)
                    or else raise Program_Error) and then
                 (Length (Container) <= Count_Type'Last - Count
                    or else raise Constraint_Error),
        Post => Length (Container)'Old + Count = Length (Container)
                and then Has_Element (Container, Position);
```

{*AI95-00302-03*} {*AI05-0257-1*} {*AI12-0112-1*} Insert inserts Count new elements prior to the element designated by Before. If Before equals No_Element, the new elements are inserted after the last node (if any). The new elements are initialized by default (see 3.3.1). Position designates the first newly-inserted element, or if Count equals 0, then Position is assigned the value of Before. Any exception raised during allocation of internal storage is propagated, and Container is not modified. 94/5

```
{AI12-0112-1} procedure Prepend (Container : in out List;                     95/5
                  New_Item  : in     Element_Type;
                  Count     : in     Count_Type := 1)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error) and then
                 (Length (Container) <= Count_Type'Last - Count
                    or else raise Constraint_Error),
        Post => Length (Container)'Old + Count = Length (Container);
```

{*AI95-00302-03*} Equivalent to Insert (Container, First (Container), New_Item, Count). 96/2

```
{AI12-0112-1} {AI12-0400-1} procedure Append (Container : in out List;        97/5
                  New_Item  : in     Element_Type;
                  Count     : in     Count_Type)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error) and then
                 (Length (Container) <= Count_Type'Last - Count
                    or else raise Constraint_Error),
        Post => Length (Container)'Old + Count = Length (Container);
```

{*AI95-00302-03*} Equivalent to Insert (Container, No_Element, New_Item, Count). 98/2

98.1/5
```
{AI12-0391-1} {AI12-0400-1} procedure Append (Container : in out List;
                  New_Item  : in     Element_Type)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error) and then
                 (Length (Container) <= Count_Type'Last - 1
                    or else raise Constraint_Error),
        Post => Length (Container)'Old + 1 = Length (Container);
```

98.2/5 {*AI12-0391-1*} Equivalent to Insert (Container, No_Element, New_Item, 1).

99/5
```
procedure Delete (Container : in out List;
                  Position  : in out Cursor;
                  Count     : in     Count_Type := 1)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error) and then
                 (Position /= No_Element
                    or else raise Constraint_Error) and then
                 (Has_Element (Container, Position)
                    or else raise Program_Error),
        Post => Length (Container)'Old - Count <= Length (Container)
                and then Position = No_Element;
```

100/5 {*AI95-00302-03*} {*AI05-0264-1*} {*AI12-0112-1*} Delete removes (from Container) Count elements starting at the element designated by Position (or all of the elements starting at Position if there are fewer than Count elements starting at Position). Finally, Position is set to No_Element.

101/5
```
{AI12-0112-1} procedure Delete_First (Container : in out List;
                       Count     : in     Count_Type := 1)
   with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error,
        Post => Length (Container)'Old - Count <= Length (Container);
```

102/3 {*AI95-00302-03*} {*AI05-0021-1*} If Length (Container) <= Count, then Delete_First is equivalent to Clear (Container). Otherwise, it removes the first Count nodes from Container.

103/5
```
{AI12-0112-1} procedure Delete_Last (Container : in out List;
                      Count     : in     Count_Type := 1)
   with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error,
        Post => Length (Container)'Old - Count <= Length (Container);
```

104/3 {*AI95-00302-03*} {*AI05-0264-1*} If Length (Container) <= Count, then Delete_Last is equivalent to Clear (Container). Otherwise, it removes the last Count nodes from Container.

105/5
```
{AI12-0112-1} procedure Reverse_Elements (Container : in out List)
   with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error;
```

106/2 {*AI95-00302-03*} Reorders the elements of Container in reverse order.

106.a/2 **Discussion:** Unlike the similar routine for a vector, elements should not be copied; rather, the nodes should be exchanged. Cursors are expected to reference the same elements afterwards.

107/5
```
procedure Swap (Container : in out List;
                I, J      : in     Cursor)
   with Pre  => (not Tampering_With_Elements_Prohibited (Container)
                    or else raise Program_Error) and then
                 (I /= No_Element or else Constraint_Error) and then
                 (J /= No_Element or else Constraint_Error) and then
                 (Has_Element (Container, I)
                    or else raise Program_Error) and then
                 (Has_Element (Container, J)
                    or else raise Program_Error);
```

108/5 {*AI95-00302-03*} {*AI12-0112-1*} Swap exchanges the values of the elements designated by I and J.

Ramification: After a call to Swap, I designates the element value previously designated by J, and J designates the element value previously designated by I. The cursors do not become ambiguous from this operation. 108.a/2

To be honest: The implementation is not required to actually copy the elements if it can do the swap some other way. But it is allowed to copy the elements if needed. 108.b/2

```
procedure Swap_Links (Container : in out List;                                    109/5
                      I, J      : in     Cursor)
   with Pre  => (not Tampering_With_Elements_Prohibited (Container)
                    or else raise Program_Error) and then
                 (I /= No_Element or else Constraint_Error) and then
                 (J /= No_Element or else Constraint_Error) and then
                 (Has_Element (Container, I)
                    or else raise Program_Error) and then
                 (Has_Element (Container, J)
                    or else raise Program_Error);
```

{*AI95-00302-03*} {*AI12-0112-1*} Swap_Links exchanges the nodes designated by I and J. 110/5

Ramification: Unlike Swap, this exchanges the nodes, not the elements. No copying is performed. I and J designate the same elements after this call as they did before it. This operation can provide better performance than Swap if the element size is large. 110.a/2

```
procedure Splice (Target   : in out List;                                         111/5
                  Before   : in     Cursor;
                  Source   : in out List)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Target)
                    or else raise Program_Error) and then
                 (not Tampering_With_Cursors_Prohibited (Source)
                    or else raise Program_Error) and then
                 (Before = No_Element or else
                  Has_Element (Target, Before)
                    or else raise Program_Error) and then
                 (Target'Has_Same_Storage (Source) or else
                  Length (Target) <= Count_Type'Last - Length (Source)
                    or else raise Constraint_Error),
        Post => (if not Target'Has_Same_Storage (Source) then
                    (declare
                       Result_Length : constant Count_Type :=
                          Length (Source)'Old + Length (Target)'Old;
                     begin
                       Length (Source) = 0 and then
                       Length (Target) = Result_Length));
```

{*AI95-00302-03*} {*AI12-0112-1*} If Source denotes the same object as Target, the operation has no effect. Otherwise, Splice reorders elements such that they are removed from Source and moved to Target, immediately prior to Before. If Before equals No_Element, the nodes of Source are spliced after the last node of Target. 112/5

113/5
```
procedure Splice (Target   : in out List;
                  Before   : in     Cursor;
                  Source   : in out List;
                  Position : in out Cursor)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Target)
                   or else raise Program_Error) and then
                (not Tampering_With_Cursors_Prohibited (Source)
                   or else raise Program_Error) and then
                (Position /= No_Element
                   or else raise Constraint_Error) and then
                (Has_Element (Source, Position)
                   or else raise Program_Error) and then
                (Before = No_Element or else
                 Has_Element (Target, Before)
                   or else raise Program_Error) and then
                (Target'Has_Same_Storage (Source) or else
                 Length (Target) <= Count_Type'Last - 1
                   or else raise Constraint_Error),
        Post => (declare
                   Org_Target_Length : constant Count_Type :=
                      Length (Target)'Old;
                   Org_Source_Length : constant Count_Type :=
                      Length (Source)'Old;
                 begin
                    (if Target'Has_Same_Storage (Source) then
                        Position = Position'Old
                     else Length (Source) = Org_Source_Length - 1 and then
                        Length (Target) = Org_Target_Length + 1 and then
                        Has_Element (Target, Position)));
```

114/5 {*AI95-00302-03*} {*AI05-0264-1*} {*AI12-0112-1*} If Source denotes the same object as Target, then there is no effect if Position equals Before, else the element designated by Position is moved immediately prior to Before, or, if Before equals No_Element, after the last element. Otherwise, the element designated by Position is removed from Source and moved to Target, immediately prior to Before, or, if Before equals No_Element, after the last element of Target. Position is updated to represent an element in Target.

114.a/2 **Ramification:** If Source is the same as Target, and Position = Before, or Next(Position) = Before, Splice has no effect, as the element does not have to move to meet the postcondition.

115/5
```
procedure Splice (Container: in out List;
                  Before   : in     Cursor;
                  Position : in     Cursor)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                   or else raise Program_Error) and then
                (Position /= No_Element
                   or else raise Constraint_Error) and then
                (Has_Element (Container, Position)
                   or else raise Program_Error) and then
                (Before = No_Element or else
                 Has_Element (Container, Before)
                   or else raise Program_Error),
        Post =>  Length (Container) = Length (Container)'Old;
```

116/5 {*AI95-00302-03*} {*AI05-0264-1*} {*AI12-0112-1*} If Position equals Before there is no effect. Otherwise, the element designated by Position is moved immediately prior to Before, or, if Before equals No_Element, after the last element.

117/5
```
{AI12-0112-1} function First (Container : List) return Cursor
   with Nonblocking, Global => null, Use_Formal => null,
        Post => (if not Is_Empty (Container)
                 then Has_Element (Container, First'Result)
                 else First'Result = No_Element);
```

118/2 {*AI95-00302-03*} If Container is empty, First returns No_Element. Otherwise, it returns a cursor that designates the first node in Container.

```
{AI12-0112-1} function First_Element (Container : List)                                    119/5
   return Element_Type
   with Pre => (not Is_Empty (Container)
                   or else raise Constraint_Error);
```

{*AI95-00302-03*} Equivalent to Element (Container, First_Index (Container)). 120/2

```
{AI12-0112-1} function Last (Container : List) return Cursor                               121/5
   with Nonblocking, Global => null, Use_Formal => null,
        Post => (if not Is_Empty (Container)
                 then Has_Element (Container, Last'Result)
                 else Last'Result = No_Element);
```

{*AI95-00302-03*} If Container is empty, Last returns No_Element. Otherwise, it returns a cursor that designates the last node in Container. 122/2

```
{AI12-0112-1} function Last_Element (Container : List)                                     123/5
   return Element_Type
   with Pre => (not Is_Empty (Container)
                   or else raise Constraint_Error);
```

{*AI95-00302-03*} Equivalent to Element (Last (Container)). 124/2

```
{AI12-0112-1} function Next (Position : Cursor) return Cursor                              125/5
   with Nonblocking, Global => in all, Use_Formal => null,
        Post => (if Position = No_Element then Next'Result = No_Element);
```

{*AI95-00302-03*} If Position equals No_Element or designates the last element of the container, then Next returns the value No_Element. Otherwise, it returns a cursor that designates the successor of the element designated by Position. 126/2

```
function Next (Container : List;                                                           126.1/5
               Position  : Cursor) return Cursor
   with Nonblocking, Global => null, Use_Formal => null,
        Pre  => Position = No_Element or else
                Has_Element (Container, Position)
                   or else raise Program_Error,
        Post => (if Position = No_Element then Next'Result = No_Element
                 elsif Next'Result = No_Element then
                   Position = Last (Container)
                 else Has_Element (Container, Next'Result));
```

{*AI12-0112-1*} Returns a cursor designating the successor of the element designated by Position in Container. 126.2/5

```
{AI12-0112-1} function Previous (Position : Cursor) return Cursor                          127/5
   with Nonblocking, Global => in all, Use_Formal => null,
        Post => (if Position = No_Element then
                   Previous'Result = No_Element);
```

{*AI95-00302-03*} If Position equals No_Element or designates the first element of the container, then Previous returns the value No_Element. Otherwise, it returns a cursor that designates the predecessor of the element designated by Position. 128/2

128.1/5
```
function Previous (Container : List;
                   Position : Cursor) return Cursor
   with Nonblocking, Global => null, Use_Formal => null,
        Pre  => Position = No_Element or else
                Has_Element (Container, Position)
                   or else raise Program_Error,
        Post => (if Position = No_Element then
                   Previous'Result = No_Element
                 elsif Previous'Result = No_Element then
                   Position = First (Container)
                 else Has_Element (Container, Previous'Result));
```

128.2/5 {*AI12-0112-1*} Returns a cursor designating the predecessor of the element designated by Position in Container, if any.

129/5
```
{AI12-0112-1} procedure Next (Position : in out Cursor)
   with Nonblocking, Global => in all, Use_Formal => null;
```

130/2 {*AI95-00302-03*} Equivalent to Position := Next (Position).

130.1/5
```
procedure Next (Container : in     List;
                Position  : in out Cursor)
   with Nonblocking, Global => null, Use_Formal => null,
        Pre  => Position = No_Element or else
                Has_Element (Container, Position)
                   or else raise Program_Error,
        Post => (if Position /= No_Element
                 then Has_Element (Container, Position));
```

130.2/5 {*AI12-0112-1*} Equivalent to Position := Next (Container, Position).

131/5
```
{AI12-0112-1} procedure Previous (Position : in out Cursor)
   with Nonblocking, Global => in all, Use_Formal => null;
```

132/2 {*AI95-00302-03*} Equivalent to Position := Previous (Position).

132.1/5
```
procedure Previous (Container : in     List;
                    Position  : in out Cursor)
   with Nonblocking, Global => null, Use_Formal => null,
        Pre  => Position = No_Element or else
                Has_Element (Container, Position)
                   or else raise Program_Error,
        Post => (if Position /= No_Element
                 then Has_Element (Container, Position));
```

132.2/5 {*AI12-0112-1*} Equivalent to Position := Previous (Container, Position).

133/5
```
function Find (Container : List;
               Item      : Element_Type;
               Position  : Cursor := No_Element)
   return Cursor
   with Pre  => Position = No_Element or else
                Has_Element (Container, Position)
                   or else raise Program_Error,
        Post => (if Find'Result /= No_Element
                 then Has_Element (Container, Find'Result));
```

134/5 {*AI95-00302-03*} {*AI12-0112-1*} Find searches the elements of Container for an element equal to Item (using the generic formal equality operator). The search starts at the element designated by Position, or at the first element if Position equals No_Element. It proceeds towards Last (Container). If no equal element is found, then Find returns No_Element. Otherwise, it returns a cursor designating the first equal element encountered.

```
function Reverse_Find (Container : List;                                    135/5
                       Item      : Element_Type;
                       Position  : Cursor := No_Element)
   return Cursor
   with Pre  => Position = No_Element or else
                Has_Element (Container, Position)
                   or else raise Program_Error,
        Post => (if Reverse_Find'Result /= No_Element
                 then Has_Element (Container, Reverse_Find'Result));
```

{*AI95-00302-03*} {*AI12-0112-1*} Find searches the elements of Container for an element equal to Item (using the generic formal equality operator). The search starts at the element designated by Position, or at the last element if Position equals No_Element. It proceeds towards First (Container). If no equal element is found, then Reverse_Find returns No_Element. Otherwise, it returns a cursor designating the first equal element encountered. 136/5

```
function Contains (Container : List;                                        137/2
                   Item      : Element_Type) return Boolean;
```

{*AI95-00302-03*} Equivalent to Find (Container, Item) /= No_Element. 138/2

Paragraphs 139 and 140 were moved above.

```
{AI12-0112-1} procedure Iterate                                             141/5
  (Container : in List;
   Process   : not null access procedure (Position : in Cursor))
   with Allows_Exit;
```

{*AI95-00302-03*} {*AI05-0265-1*} Iterate calls Process.**all** with a cursor that designates each node in Container, starting with the first node and moving the cursor as per the Next function. Tampering with the cursors of Container is prohibited during the execution of a call on Process.**all**. Any exception raised by Process.**all** is propagated. 142/3

Implementation Note: The purpose of the tamper with cursors check is to prevent erroneous execution from the Position parameter of Process.**all** becoming invalid. This check takes place when the operations that tamper with the cursors of the container are called. The check cannot be made later (say in the body of Iterate), because that could cause the Position cursor to be invalid and potentially cause execution to become erroneous -- defeating the purpose of the check. 142.a/2

See Iterate for vectors (A.18.2) for a suggested implementation of the check. 142.b/2

```
{AI12-0112-1} procedure Reverse_Iterate                                     143/5
  (Container : in List;
   Process   : not null access procedure (Position : in Cursor))
   with Allows_Exit;
```

{*AI95-00302-03*} {*AI05-0212-1*} Iterates over the nodes in Container as per procedure Iterate, except that elements are traversed in reverse order, starting with the last node and moving the cursor as per the Previous function. 144/3

```
{AI12-0112-1} {AI12-0266-1} function Iterate (Container : in List)          144.1/5
   return List_Iterator_Interfaces.Parallel_Reversible_Iterator'Class
   with Post   => Tampering_With_Cursors_Prohibited (Container);
```

{*AI05-0212-1*} {*AI05-0265-1*} {*AI05-0269-1*} {*AI12-0266-1*} Iterate returns an iterator object (see 5.5.1) that will generate a value for a loop parameter (see 5.5.2) designating each node in Container, starting with the first node and moving the cursor as per the Next function when used as a forward iterator, and starting with the last node and moving the cursor as per the Previous function when used as a reverse iterator, and processing all nodes concurrently when used as a parallel iterator. Tampering with the cursors of Container is prohibited while the iterator object exists (in particular, in the sequence_of_statements of the loop_statement whose iterator_specification denotes this object). The iterator object needs finalization. 144.2/5

144.3/5
```
function Iterate (Container : in List; Start : in Cursor)
   return List_Iterator_Interfaces.Reversible_Iterator'Class
   with Pre    => (Start /= No_Element
                      or else raise Constraint_Error) and then
                  (Has_Element (Container, Start)
                      or else raise Program_Error),
        Post   => Tampering_With_Cursors_Prohibited (Container);
```

144.4/5 {*AI05-0212-1*} {*AI05-0262-1*} {*AI05-0265-1*} {*AI05-0269-1*} {*AI12-0112-1*} Iterate returns a reversible iterator object (see 5.5.1) that will generate a value for a loop parameter (see 5.5.2) designating each node in Container, starting with the node designated by Start and moving the cursor as per the Next function when used as a forward iterator, or moving the cursor as per the Previous function when used as a reverse iterator. Tampering with the cursors of Container is prohibited while the iterator object exists (in particular, in the sequence_of_statements of the loop_statement whose iterator_specification denotes this object). The iterator object needs finalization.

144.a/3 **Discussion:** Exits are allowed from the loops created using the iterator objects. In particular, to stop the iteration at a particular cursor, just add

144.b/3
```
exit when Cur = Stop;
```

144.c/3 in the body of the loop (assuming that `Cur` is the loop parameter and `Stop` is the cursor that you want to stop at).

145/3 {*AI05-0044-1*} {*AI05-0262-1*} The actual function for the generic formal function "<" of Generic_Sorting is expected to return the same value each time it is called with a particular pair of element values. It should define a strict weak ordering relationship (see A.18); it should not modify Container. If the actual for "<" behaves in some other manner, the behavior of the subprograms of Generic_Sorting are unspecified. The number of times the subprograms of Generic_Sorting call "<" is unspecified.

146/2
```
function Is_Sorted (Container : List) return Boolean;
```

147/2 {*AI95-00302-03*} Returns True if the elements are sorted smallest first as determined by the generic formal "<" operator; otherwise, Is_Sorted returns False. Any exception raised during evaluation of "<" is propagated.

148/5
```
{AI12-0112-1} procedure Sort (Container : in out List)
   with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error;
```

149/2 {*AI95-00302-03*} Reorders the nodes of Container such that the elements are sorted smallest first as determined by the generic formal "<" operator provided. The sort is stable. Any exception raised during evaluation of "<" is propagated.

149.a/2 **Ramification:** Unlike array sorts, we do require stable sorts here. That's because algorithms in the merge sort family (as described by Knuth) can be both fast and stable. Such sorts use the extra memory as offered by the links to provide better performance.

149.b/2 Note that list sorts never copy elements; it is the nodes, not the elements, that are reordered.

```
procedure Merge (Target  : in out List;                                    150/5
                 Source  : in out List)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Target)
                    or else raise Program_Error) and then
                 (not Tampering_With_Elements_Prohibited (Source)
                    or else raise Program_Error) and then
                 (Length (Target) <= Count_Type'Last - Length (Source)
                    or else raise Constraint_Error) and then
                 ((Length (Source) = 0 or else
                    not Target'Has_Same_Storage (Source))
                    or else raise Constraint_Error),
        Post => (declare
                    Result_Length : constant Count_Type :=
                       Length (Source)'Old + Length (Target)'Old;
                 begin
                    (Length (Source) = 0 and then
                     Length (Target) = Result_Length));
```

{*AI95-00302-03*} {*AI05-0021-1*} {*AI12-0112-1*} Merge removes elements from Source and inserts them into Target; afterwards, Target contains the union of the elements that were initially in Source and Target; Source is left empty. If Target and Source are initially sorted smallest first, then Target is ordered smallest first as determined by the generic formal "<" operator; otherwise, the order of elements in Target is unspecified. Any exception raised during evaluation of "<" is propagated. 151/5

Ramification: It is a bounded error if either of the lists is unsorted, see below. The bounded error can be recovered by sorting Target after the merge call, or the lists can be pretested with Is_Sorted. 151.a/2

{*AI12-0111-1*} The nested package Doubly_Linked_Lists.Stable provides a type Stable.List that represents a *stable* list, which is one that cannot grow and shrink. Such a list can be created by calling the Copy function, or by establishing a *stabilized view* of an ordinary list. 151.1/5

{*AI12-0111-1*} The subprograms of package Containers.Doubly_Linked_Lists that have a parameter or result of type List are included in the nested package Stable with the same specification, except that the following are omitted: 151.2/5

Tampering_With_Cursors_Prohibited, Tampering_With_Elements_Prohibited, Assign, Move, Insert, Append, Prepend, Clear, Delete, Delete_First, Delete_Last, Splice, Swap_Links, and Reverse_Elements 151.3/5

Ramification: The names List and Cursor mean the types declared in the nested package in these subprogram specifications. 151.a.1/5

Reason: The omitted routines are those that tamper with cursors or elements (or test that state). The model is that it is impossible to tamper with cursors or elements of a stable view since no such operations are included. Thus tampering checks are not needed for a stable view, and we omit the operations associated with those checks. 151.a.2/5

The Generic_Sorting generic is omitted entirely, as only function Is_Sorting does not tamper with cursors. It isn't useful enough by itself to include. 151.a.3/5

{*AI12-0111-1*} The operations of this package are equivalent to those for ordinary lists, except that the calls to Tampering_With_Cursors_Prohibited and Tampering_With_Elements_Prohibited that occur in preconditions are replaced by False, and any that occur in postconditions are replaced by True. 151.4/5

{*AI12-0111-1*} {*AI12-0439-1*} If a stable list is declared with the Base discriminant designating a pre-existing ordinary list, the stable list represents a stabilized view of the underlying ordinary list, and any operation on the stable list is reflected on the underlying ordinary list. While a stabilized view exists, any operation that tampers with elements performed on the underlying list is prohibited. The finalization of a stable list that provides such a view removes this restriction on the underlying ordinary list [(though some other restriction can exist due to other concurrent iterations or stabilized views)]. 151.5/5

151.6/5 {*AI12-0111-1*} {*AI12-0438-1*} If a stable list is declared without specifying Base, the object is necessarily initialized. The initializing expression of the stable list, [typically a call on Copy], determines the Length of the list. The Length of a stable list never changes after initialization.

151.b/5 **Proof:** {*AI12-0438-1*} Initialization is required as the type is indefinite, see 3.3.1.

Bounded (Run-Time) Errors

152/2 {*AI95-00302-03*} Calling Merge in an instance of Generic_Sorting with either Source or Target not ordered smallest first using the provided generic formal "<" operator is a bounded error. Either Program_Error is raised after Target is updated as described for Merge, or the operation works as defined.

152.1/3 {*AI05-0022-1*} {*AI05-0248-1*} It is a bounded error for the actual function associated with a generic formal subprogram, when called as part of an operation of this package, to tamper with elements of any List parameter of the operation. Either Program_Error is raised, or the operation works as defined on the value of the List either prior to, or subsequent to, some or all of the modifications to the List.

152.2/3 {*AI05-0027-1*} It is a bounded error to call any subprogram declared in the visible part of Containers.Doubly_Linked_Lists when the associated container has been finalized. If the operation takes Container as an **in out** parameter, then it raises Constraint_Error or Program_Error. Otherwise, the operation either proceeds as it would for an empty container, or it raises Constraint_Error or Program_Error.

Erroneous Execution

153/2 {*AI95-00302-03*} A Cursor value is *invalid* if any of the following have occurred since it was created:

154/2 • The list that contains the element it designates has been finalized;

154.1/3 • {*AI05-0160-1*} The list that contains the element it designates has been used as the Target of a call to Assign, or as the target of an assignment_statement;

155/2 • [The list that contains the element it designates has been used as the Source or Target of a call to Move;] or

155.a/3 **Proof:** {*AI05-0001-1*} Move has been reworded in terms of Assign and Clear, which are covered by other bullets, so this text is redundant.

156/3 • {*AI05-0160-1*} {*AI05-0262-1*} The element it designates has been removed from the list that previously contained the element.

156.a/3 **To be honest:** {*AI05-0160-1*} The cursor modified by the four parameter Splice is not invalid, even though the element it designates has been removed from the source list, because that cursor has been modified to designate that element in the target list – the cursor no longer designates an element in the source list.

156.b/3 **Ramification:** {*AI05-0160-1*} This can happen directly via calls to Delete, Delete_Last, Clear, Splice with a Source parameter, and Merge; and indirectly via calls to Delete_First, Assign, and Move.

157/2 {*AI95-00302-03*} The result of "=" or Has_Element is unspecified if it is called with an invalid cursor parameter. Execution is erroneous if any other subprogram declared in Containers.Doubly_Linked_Lists is called with an invalid cursor parameter.

157.a/2 **Discussion:** The list above is intended to be exhaustive. In other cases, a cursor value continues to designate its original element. For instance, cursor values survive the insertion and deletion of other nodes.

157.b/2 While it is possible to check for these cases, in many cases the overhead necessary to make the check is substantial in time or space. Implementations are encouraged to check for as many of these cases as possible and raise Program_Error if detected.

157.1/3 {*AI05-0212-1*} Execution is erroneous if the list associated with the result of a call to Reference or Constant_Reference is finalized before the result object returned by the call to Reference or Constant_Reference is finalized.

Reason: Each object of Reference_Type and Constant_Reference_Type probably contains some reference to the originating container. If that container is prematurely finalized (which is only possible via Unchecked_Deallocation, as accessibility checks prevent passing a container to Reference that will not live as long as the result), the finalization of the object of Reference_Type will try to access a nonexistent object. This is a normal case of a dangling pointer created by Unchecked_Deallocation; we have to explicitly mention it here as the pointer in question is not visible in the specification of the type. (This is the same reason we have to say this for invalid cursors.) 157.c/3

Implementation Requirements

{*AI95-00302-03*} {*AI12-0437-1*} No storage associated with a doubly-linked list object shall be lost upon assignment or scope exit. 158/5

{*AI95-00302-03*} {*AI05-0262-1*} The execution of an assignment_statement for a list shall have the effect of copying the elements from the source list object to the target list object and changing the length of the target object to that of the source object. 159/3

Implementation Note: {*AI05-0298-1*} An assignment of a List is a "deep" copy; that is the elements are copied as well as the data structures. We say "effect of" in order to allow the implementation to avoid copying elements immediately if it wishes. For instance, an implementation that avoided copying until one of the containers is modified would be allowed. (Note that this implementation would require care, see A.18.2 for more.) 159.a/3

Implementation Advice

{*AI95-00302-03*} Containers.Doubly_Linked_Lists should be implemented similarly to a linked list. In particular, if *N* is the length of a list, then the worst-case time complexity of Element, Insert with Count=1, and Delete with Count=1 should be *O*(log *N*). 160/2

Implementation Advice: The worst-case time complexity of Element, Insert with Count=1, and Delete with Count=1 for Containers.Doubly_Linked_Lists should be *O*(log *N*). 160.a/2

Reason: We do not mean to overly constrain implementation strategies here. However, it is important for portability that the performance of large containers has roughly the same factors on different implementations. If a program is moved to an implementation that takes *O*(*N*) time to access elements, that program could be unusable when the lists are large. We allow *O*(log *N*) access because the proportionality constant and caching effects are likely to be larger than the log factor, and we don't want to discourage innovative implementations. 160.b/2

{*AI95-00302-03*} The worst-case time complexity of a call on procedure Sort of an instance of Containers.Doubly_Linked_Lists.Generic_Sorting should be *O*(*N***2), and the average time complexity should be better than *O*(*N***2). 161/2

Implementation Advice: A call on procedure Sort of an instance of Containers.Doubly_Linked_Lists.Generic_Sorting should have an average time complexity better than *O*(*N***2) and worst case no worse than *O*(*N***2). 161.a/2

Ramification: In other words, we're requiring the use of a better than *O*(*N***2) sorting algorithm, such as Quicksort. No bubble sorts allowed! 161.b/2

{*AI95-00302-03*} Move should not copy elements, and should minimize copying of internal data structures. 162/2

Implementation Advice: Containers.Doubly_Linked_Lists.Move should not copy elements, and should minimize copying of internal data structures. 162.a/2

Implementation Note: Usually that can be accomplished simply by moving the pointer(s) to the internal data structures from the Source container to the Target container. 162.b/2

{*AI95-00302-03*} If an exception is propagated from a list operation, no storage should be lost, nor any elements removed from a list unless specified by the operation. 163/2

Implementation Advice: If an exception is propagated from a list operation, no storage should be lost, nor any elements removed from a list unless specified by the operation. 163.a/2

Reason: This is important so that programs can recover from errors. But we don't want to require heroic efforts, so we just require documentation of cases where this can't be accomplished. 163.b/2

NOTE {*AI95-00302-03*} {*AI12-0442-1*} Sorting a list never copies elements, and is a stable sort (equal elements remain in the original order). This is different than sorting an array or vector, which will often copy elements, and hence is probably not a stable sort. 164/5

Extensions to Ada 95

164.a/2 {*AI95-00302-03*} The generic package Containers.Doubly_Linked_Lists is new.

Inconsistencies With Ada 2005

164.b/3 {*AI05-0248-1*} {*AI05-0257-1*} **Correction:** The Insert versions that return a Position parameter are now defined to return Position = Before if Count = 0. This was unspecified for Ada 2005; so this will only be inconsistent if an implementation did something else and a program depended on that something else — this should be very rare.

Incompatibilities With Ada 2005

164.c/3 {*AI05-0001-1*} Subprograms Assign and Copy are added to Containers.Doubly_Linked_Lists. If an instance of Containers.Doubly_Linked_Lists is referenced in a use_clause, and an entity *E* with the same defining_identifier as a new entity in Containers.Doubly_Linked_Lists is defined in a package that is also referenced in a use_clause, the entity *E* may no longer be use-visible, resulting in errors. This should be rare and is easily fixed if it does occur.

Extensions to Ada 2005

164.d/3 {*AI05-0212-1*} Added iterator, reference, and indexing support to make list containers more convenient to use.

Wording Changes from Ada 2005

164.e/3 {*AI05-0001-1*} Generalized the definition of Move. Specified which elements are read/written by stream attributes.

164.f/3 {*AI05-0022-1*} **Correction:** Added a Bounded (Run-Time) Error to cover tampering by generic actual subprograms.

164.g/3 {*AI05-0027-1*} **Correction:** Added a Bounded (Run-Time) Error to cover access to finalized list containers.

164.h/3 {*AI05-0044-1*} **Correction:** Redefined "<" actuals to require a strict weak ordering; the old definition allowed indeterminant comparisons that would not have worked in a container.

164.i/3 {*AI05-0084-1*} **Correction:** Added a pragma Remote_Types so that containers can be used in distributed programs.

164.j/3 {*AI05-0160-1*} **Correction:** Revised the definition of invalid cursors to cover missing (and new) cases.

164.k/3 {*AI05-0257-1*} **Correction:** Added missing wording to describe the Position after Inserting 0 elements.

164.l/3 {*AI05-0265-1*} **Correction:** Defined when a container prohibits tampering in order to more clearly define where the check is made and the exception raised.

Inconsistencies With Ada 2012

164.m/5 {*AI12-0111-1*} **Correction:** Tampering with elements is now defined to be equivalent to tampering with cursors for ordinary containers. If a program requires tampering detection to work, it might fail in Ada 2022. Needless to say, this shouldn't happen outside of test programs. See Inconsistencies With Ada 2012 in A.18.2 for more details.

Incompatibilities With Ada 2012

164.n/5 {*AI12-0111-1*} {*AI12-0112-1*} {*AI12-0339-1*} {*AI12-0391-1*} A number of new subprograms, types, and even a nested package were added to Containers.Doubly_Linked_Lists to better support contracts and stable views. Therefore, a use clause conflict is possible; see the introduction of Annex A for more on this topic.

Extensions to Ada 2012

164.o/5 {*AI12-0196-1*} Replace_Element is now defined such that it can be used concurrently so long as it operates on different elements. This allows some container operations to be used in parallel without separate synchronization.

164.p/5 {*AI12-0212-1*} {*AI12-0391-1*} Lists now support positional container aggregates, so aggregate syntax can be used to create Lists.

164.q/5 {*AI12-0266-1*} The iterator for the entire container now can return a parallel iterator which can be used to process the container in parallel.

Wording Changes from Ada 2012

164.r/4 {*AI12-0110-1*} **Corrigendum:** Clarified that tampering checks precede all other checks made by a subprogram (but come after those associated with the call).

164.s/5 {*AI12-0112-1*} Added contracts to this container. This includes describing some of the semantics with pre- and postconditions, rather than English text. Note that the preconditions can be Suppressed (see 11.5).

{*AI12-0400-1*} **Correction:** Split the Append routine into two routines rather than having a single routine with a default parameter, in order that a routine with the appropriate profile for the Aggregate aspect exists. This change should not change the behavior of any existing code. 164.t/5

A.18.4 Maps

{*AI95-00302-03*} The language-defined generic packages Containers.Hashed_Maps and Containers.Ordered_Maps provide private types Map and Cursor, and a set of operations for each type. A map container allows an arbitrary type to be used as a key to find the element associated with that key. A hashed map uses a hash function to organize the keys, while an ordered map orders the keys per a specified relation. 1/2

{*AI95-00302-03*} {*AI05-0299-1*} This subclause describes the declarations that are common to both kinds of maps. See A.18.5 for a description of the semantics specific to Containers.Hashed_Maps and A.18.6 for a description of the semantics specific to Containers.Ordered_Maps. 2/3

Static Semantics

{*AI95-00302-03*} The actual function for the generic formal function "=" on Element_Type values is expected to define a reflexive and symmetric relationship and return the same result value each time it is called with a particular pair of values. If it behaves in some other manner, the function "=" on map values returns an unspecified value. The exact arguments and number of calls of this generic formal function by the function "=" on map values are unspecified. 3/2

Ramification: If the actual function for "=" is not symmetric and consistent, the result returned by "=" for Map objects cannot be predicted. The implementation is not required to protect against "=" raising an exception, or returning random results, or any other “bad” behavior. And it can call "=" in whatever manner makes sense. But note that only the result of "=" for Map objects is unspecified; other subprograms are not allowed to break if "=" is bad (they aren't expected to use "="). 3.a/2

{*AI95-00302-03*} The type Map is used to represent maps. The type Map needs finalization (see 7.6). 4/2

{*AI95-00302-03*} A map contains pairs of keys and elements, called *nodes*. Map cursors designate nodes, but also can be thought of as designating an element (the element contained in the node) for consistency with the other containers. There exists an equivalence relation on keys, whose definition is different for hashed maps and ordered maps. A map never contains two or more nodes with equivalent keys. The *length* of a map is the number of nodes it contains. 5/2

{*AI95-00302-03*} Each nonempty map has two particular nodes called the *first node* and the *last node* (which may be the same). Each node except for the last node has a *successor node*. If there are no other intervening operations, starting with the first node and repeatedly going to the successor node will visit each node in the map exactly once until the last node is reached. The exact definition of these terms is different for hashed maps and ordered maps. 6/2

{*AI95-00302-03*} {*AI12-0111-1*} {*AI12-0112-1*} [Some operations check for “tampering with cursors” of a container because they depend on the set of elements of the container remaining constant, and others check for “tampering with elements” of a container because they depend on elements of the container not being replaced.] When tampering with cursors is *prohibited* for a particular map object *M*, Program_Error is propagated by the finalization of *M*[, as well as by a call that passes *M* to certain of the operations of this package, as indicated by the precondition of such an operation]. Similarly, when tampering with elements is *prohibited* for *M*, Program_Error is propagated by a call that passes *M* to certain of the other operations of this package, as indicated by the precondition of such an operation. 7/5

Paragraphs 8 through 15 are removed as preconditions now describe these rules.

Ramification: We don't need to explicitly mention assignment_statement, because that finalizes the target object as part of the operation, and finalization of an object is already defined as tampering with cursors. 10.a/3

16/2 {*AI95-00302-03*} Empty_Map represents the empty Map object. It has a length of 0. If an object of type Map is not otherwise initialized, it is initialized to the same value as Empty_Map.

17/2 {*AI95-00302-03*} No_Element represents a cursor that designates no node. If an object of type Cursor is not otherwise initialized, it is initialized to the same value as No_Element.

18/5 {*AI95-00302-03*} {*AI12-0434-1*} The primitive "=" operator for type Cursor returns True if both cursors are No_Element, or designate the same element in the same container.

18.a/5 **To be honest:** {*AI12-0434-1*} "The primitive "=" operator" is the one with two parameters of type Cursor which returns Boolean. We're not talking about some other (hidden) primitive function named "=".

19/2 {*AI95-00302-03*} Execution of the default implementation of the Input, Output, Read, or Write attribute of type Cursor raises Program_Error.

19.a/2 **Reason:** A cursor will probably be implemented in terms of one or more access values, and the effects of streaming access values is unspecified. Rather than letting the user stream junk by accident, we mandate that streaming of cursors raise Program_Error by default. The attributes can always be specified if there is a need to support streaming.

19.1/5 {*AI05-0001-1*} {*AI05-0262-1*} {*AI12-0437-1*} Map'Write for a Map object *M* writes Length(*M*) elements of the map to the stream. It may also write additional information about the map.

19.2/3 {*AI05-0001-1*} {*AI05-0262-1*} Map'Read reads the representation of a map from the stream, and assigns to *Item* a map with the same length and elements as was written by Map'Write.

19.b/3 **Ramification:** Streaming more elements than the container length is wrong. For implementation implications of this rule, see the Implementation Note in A.18.2.

19.3/5
```
{AI12-0112-1} function Has_Element (Position : Cursor) return Boolean
   with Nonblocking, Global => in all, Use_Formal => null;
```

19.4/3 {*AI05-0212-1*} Returns True if Position designates an element, and returns False otherwise.

19.c/3 **To be honest:** {*AI05-0005-1*} {*AI05-0212-1*} This function might not detect cursors that designate deleted elements; such cursors are invalid (see below) and the result of calling Has_Element with an invalid cursor is unspecified (but not erroneous).

19.5/5
```
{AI12-0112-1} function Has_Element (Container : Map; Position : Cursor)
   return Boolean
   with Nonblocking, Global => null, Use_Formal => null;
```

19.6/5 {*AI12-0112-1*} Returns True if Position designates an element in Container, and returns False otherwise.

19.d/5 **Ramification:** {*AI12-0112-1*} If Position is No_element, Has_Element returns False.

20/2
```
function "=" (Left, Right : Map) return Boolean;
```

21/2 {*AI95-00302-03*} If Left and Right denote the same map object, then the function returns True. If Left and Right have different lengths, then the function returns False. Otherwise, for each key *K* in Left, the function returns False if:

22/2 - a key equivalent to *K* is not present in Right; or

23/2 - the element associated with *K* in Left is not equal to the element associated with *K* in Right (using the generic formal equality operator for elements).

24/2 If the function has not returned a result after checking all of the keys, it returns True. Any exception raised during evaluation of key equivalence or element equality is propagated.

24.a/2 **Implementation Note:** This wording describes the canonical semantics. However, the order and number of calls on the formal equality function is unspecified for all of the operations that use it in this package, so an implementation can call it as many or as few times as it needs to get the correct answer. Specifically, there is no requirement to call the formal equality additional times once the answer has been determined.

```
function Tampering_With_Cursors_Prohibited                                    24.1/5
   (Container : Map) return Boolean
   with Nonblocking, Global => null, Use_Formal => null;
```

{*AI12-0112-1*} Returns True if tampering with cursors or tampering with elements is currently prohibited for Container, and returns False otherwise. 24.2/5

Reason: {*AI12-0112-1*} Prohibiting tampering with elements also needs to prohibit tampering with cursors, as deleting an element is similar to replacing it. 24.b/5

Implementation Note: {*AI12-0112-1*} Various contracts elsewhere in this specification require that this function be implemented with synchronized data. Moreover, it is possible for tampering to be prohibited by multiple operations (sequentially or in parallel). Therefore, tampering needs to be implemented with an atomic or protected counter. The counter is initialized to zero, and is incremented when tampering is prohibited, and decremented when leaving an area that prohibited tampering. Function Tampering_With_Cursors_Prohibited returns True if the counter is nonzero. (Note that any case where the result is not well-defined for one task is incorrect use of shared variables and would be erroneous by the rules of 9.10, so no special protection is needed to read the counter.) 24.c/5

```
function Tampering_With_Elements_Prohibited                                   24.3/5
   (Container : Map) return Boolean
   with Nonblocking, Global => null, Use_Formal => null;
```

{*AI12-0112-1*} Always returns False[, regardless of whether tampering with elements is prohibited]. 24.4/5

Reason: {*AI12-0111-1*} A definite element cannot change size, so we allow operations that tamper with elements even when tampering with elements is prohibited. That's not true for the indefinite containers, which is why this kind of tampering exists. 24.d/5

```
{AI12-0112-1} function Length (Container : Map) return Count_Type             25/5
   with Nonblocking, Global => null, Use_Formal => null;
```

{*AI95-00302-03*} Returns the number of nodes in Container. 26/2

```
function Is_Empty (Container : Map) return Boolean                            27/5
   with Nonblocking, Global => null, Use_Formal => null,
        Post => Is_Empty'Result = (Length (Container) = 0);
```

{*AI95-00302-03*} {*AI12-0112-1*} Returns True if Container is empty. 28/5

```
{AI12-0112-1} procedure Clear (Container : in out Map)                        29/5
   with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error,
        Post => Length (Container) = 0;
```

{*AI95-00302-03*} Removes all the nodes from Container. 30/2

```
function Key (Position : Cursor) return Key_Type                              31/5
   with Pre  => Position /= No_Element
                    or else raise Constraint_Error,
        Nonblocking, Global => in all, Use_Formal => Key_Type;
```

{*AI95-00302-03*} {*AI12-0112-1*} Key returns the key component of the node designated by Position. 32/5

```
function Key (Container : Map;                                                32.1/5
              Position : Cursor) return Key_Type
   with Pre  => (Position /= No_Element
                    or else raise Constraint_Error) and then
                (Has_Element (Container, Position)
                   or else raise Program_Error),
        Nonblocking, Global => null, Use_Formal => Key_Type;
```

{*AI12-0112-1*} Key returns the key component of the node designated by Position. 32.2/5

33/5
```
function Element (Position : Cursor) return Element_Type
   with Pre  => Position /= No_Element
                   or else raise Constraint_Error,
        Nonblocking, Global => in all, Use_Formal => Element_Type;
```

34/5 {*AI95-00302-03*} {*AI12-0112-1*} Element returns the element component of the node designated by Position.

34.1/5
```
function Element (Container : Map;
                  Position  : Cursor) return Element_Type
   with Pre  => (Position /= No_Element
                   or else raise Constraint_Error) and then
                (Has_Element (Container, Position)
                   or else raise Program_Error),
        Nonblocking, Global => null, Use_Formal => Element_Type;
```

34.2/5 {*AI12-0112-1*} Element returns the element component of the node designated by Position.

35/5
```
procedure Replace_Element (Container : in out Map;
                           Position  : in     Cursor;
                           New_item  : in     Element_Type)
   with Pre  => (not Tampering_With_Elements_Prohibited (Container)
                   or else raise Program_Error) and then
                (Position /= No_Element
                   or else raise Constraint_Error) and then
                (Has_Element (Container, Position)
                   or else raise Program_Error);
```

36/5 {*AI95-00302-03*} {*AI05-0264-1*} {*AI12-0112-1*} {*AI12-0196-1*} Replace_Element assigns New_Item to the element of the node designated by Position. For the purposes of determining whether the parameters overlap in a call to Replace_Element, the Container parameter is not considered to overlap with any object [(including itself)].

37/5
```
procedure Query_Element
  (Position : in Cursor;
   Process  : not null access procedure (Key     : in Key_Type;
                                         Element : in Element_Type))
   with Pre  => Position /= No_Element
                   or else raise Constraint_Error,
        Global => in all;
```

38/5 {*AI95-00302-03*} {*AI05-0021-1*} {*AI05-0265-1*} {*AI12-0112-1*} Query_Element calls Process.**all** with the key and element from the node designated by Position as the arguments. Tampering with the elements of the map that contains the element designated by Position is prohibited during the execution of the call on Process.**all**. Any exception raised by Process.**all** is propagated.

38.1/5
```
procedure Query_Element
  (Container : in Map;
   Position  : in Cursor;
   Process   : not null access procedure (Key     : in Key_Type;
                                          Element : in Element_Type))
  with Pre  => (Position /= No_Element
                   or else raise Constraint_Error) and then
               (Has_Element (Container, Position)
                   or else raise Program_Error);
```

38.2/5 {*AI12-0112-1*} Query_Element calls Process.**all** with the key and element from the node designated by Position as the arguments. Tampering with the elements of Container is prohibited during the execution of the call on Process.**all**. Any exception raised by Process.**all** is propagated.

```
procedure Update_Element                                                    39/5
  (Container : in out Map;
   Position  : in     Cursor;
   Process   : not null access procedure (Key     : in     Key_Type;
                                          Element : in out Element_Type))
   with Pre  => (Position /= No_Element
                    or else raise Constraint_Error) and then
                 (Has_Element (Container, Position)
                    or else raise Program_Error);
```

{*AI95-00302-03*} {*AI05-0264-1*} {*AI05-0265-1*} {*AI12-0112-1*} Update_Element calls Process.**all** with the key and element from the node designated by Position as the arguments. Tampering with the elements of Container is prohibited during the execution of the call on Process.**all**. Any exception raised by Process.**all** is propagated. 40/5

If Element_Type is unconstrained and definite, then the actual Element parameter of Process.**all** shall be unconstrained. 41/2

Ramification: This means that the elements cannot be directly allocated from the heap; it must be possible to change the discriminants of the element in place. 41.a/2

```
{AI12-0112-1} type Constant_Reference_Type                                  41.1/5
      (Element : not null access constant Element_Type) is private
   with Implicit_Dereference => Element,
        Nonblocking, Global =>in out synchronized,
        Default_Initial_Condition => (raise Program_Error);
```

```
{AI12-0112-1} type Reference_Type (Element : not null access Element_Type) is  41.2/5
private
   with Implicit_Dereference => Element,
        Nonblocking, Global => in out synchronized,
        Default_Initial_Condition => (raise Program_Error);
```

{*AI05-0212-1*} The types Constant_Reference_Type and Reference_Type need finalization. 41.3/3

This paragraph was deleted.{*AI12-0112-1*} 41.4/5

Reason: It is expected that Reference_Type (and Constant_Reference_Type) will be a controlled type, for which finalization will have some action to terminate the tampering check for the associated container. If the object is created by default, however, there is no associated container. Since this is useless, and supporting this case would take extra work, we define it to raise an exception. 41.b/3

```
{AI12-0112-1} function Constant_Reference (Container : aliased in Map;      41.5/5
                           Position  : in Cursor)
   return Constant_Reference_Type
   with Pre  => (Position /= No_Element
                    or else raise Constraint_Error) and then
                 (Has_Element (Container, Position)
                    or else raise Program_Error),
        Post => Tampering_With_Cursors_Prohibited (Container),
        Nonblocking, Global => null, Use_Formal => null;
```

{*AI05-0212-1*} {*AI05-0269-1*} This function (combined with the Constant_Indexing and Implicit_Dereference aspects) provides a convenient way to gain read access to an individual element of a Map given a cursor. 41.6/3

{*AI05-0212-1*} {*AI05-0265-1*} {*AI12-0112-1*} Constant_Reference returns an object whose discriminant is an access value that designates the element designated by Position. Tampering with the elements of Container is prohibited while the object returned by Constant_Reference exists and has not been finalized. 41.7/5

41.8/5
```
{AI12-0112-1} function Reference (Container : aliased in out Map;
                                  Position  : in Cursor)
   return Reference_Type
   with Pre  => (Position /= No_Element
                    or else raise Constraint_Error) and then
                 (Has_Element (Container, Position)
                    or else raise Program_Error),
        Post => Tampering_With_Cursors_Prohibited (Container),
        Nonblocking, Global => null, Use_Formal => null;
```

41.9/3 {*AI05-0212-1*} {*AI05-0269-1*} This function (combined with the Variable_Indexing and Implicit_Dereference aspects) provides a convenient way to gain read and write access to an individual element of a Map given a cursor.

41.10/5 {*AI05-0212-1*} {*AI05-0265-1*} {*AI12-0112-1*} Reference returns an object whose discriminant is an access value that designates the element designated by Position. Tampering with the elements of Container is prohibited while the object returned by Reference exists and has not been finalized.

41.11/5
```
{AI12-0112-1} function Constant_Reference (Container : aliased in Map;
                                           Key       : in Key_Type)
   return Constant_Reference_Type
   with Pre  => Find (Container, Key) /= No_Element
                    or else raise Constraint_Error,
        Post => Tampering_With_Cursors_Prohibited (Container),
        Nonblocking, Global => null, Use_Formal => null;
```

41.12/3 {*AI05-0212-1*} {*AI05-0269-1*} This function (combined with the Constant_Indexing and Implicit_Dereference aspects) provides a convenient way to gain read access to an individual element of a map given a key value.

41.13/3 Equivalent to Constant_Reference (Container, Find (Container, Key)).

41.14/5
```
{AI12-0112-1} function Reference (Container : aliased in out Map;
                                  Key       : in Key_Type)
   return Reference_Type
   with Pre  => Find (Container, Key) /= No_Element
                    or else raise Constraint_Error,
        Post => Tampering_With_Cursors_Prohibited (Container),
        Nonblocking, Global => null, Use_Formal => null;
```

41.15/3 {*AI05-0212-1*} {*AI05-0269-1*} This function (combined with the Variable_Indexing and Implicit_Dereference aspects) provides a convenient way to gain read and write access to an individual element of a map given a key value.

41.16/3 Equivalent to Reference (Container, Find (Container, Key)).

41.17/5
```
{AI12-0112-1} procedure Assign (Target : in out Map; Source : in Map)
   with Pre  => not Tampering_With_Cursors_Prohibited (Target)
                    or else raise Program_Error,
        Post => Length (Source) = Length (Target);
```

41.18/3 {*AI05-0001-1*} {*AI05-0248-1*} If Target denotes the same object as Source, the operation has no effect. Otherwise, the key/element pairs of Source are copied to Target as for an assignment_statement assigning Source to Target.

41.c/3 **Discussion:** {*AI05-0005-1*} This routine exists for compatibility with the bounded map containers. For an unbounded map, `Assign(A, B)` and `A := B` behave identically. For a bounded map, := will raise an exception if the container capacities are different, while Assign will not raise an exception if there is enough room in the target.

```
{AI12-0112-1} procedure Move (Target : in out Map;                                    42/5
                              Source : in out Map)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Target)
                     or else raise Program_Error) and then
                 (not Tampering_With_Cursors_Prohibited (Source)
                     or else raise Program_Error),
        Post => (if not Target'Has_Same_Storage (Source) then
                     Length (Target) = Length (Source'Old) and then
                     Length (Source) = 0);
```

{*AI95-00302-03*} {*AI05-0001-1*} {*AI05-0248-1*} {*AI05-0262-1*} If Target denotes the same object as Source, then the operation has no effect. Otherwise, the operation is equivalent to Assign (Target, Source) followed by Clear (Source). 43/3

```
{AI12-0112-1} procedure Insert (Container : in out Map;                               44/5
                                Key       : in     Key_Type;
                                New_Item  : in     Element_Type;
                                Position  :    out Cursor;
                                Inserted  :    out Boolean)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                     or else raise Program_Error) and then
                 (Length (Container) <= Count_Type'Last - 1
                     or else raise Constraint_Error),
        Post => (declare
                    Original_Length : constant Count_Type :=
                       Length (Container)'Old;
                 begin
                   Has_Element (Container, Position) and then
                   (if Inserted then
                      Length (Container) = Original_Length + 1
                    else
                      Length (Container) = Original_Length));
```

{*AI95-00302-03*} Insert checks if a node with a key equivalent to Key is already present in Container. If a match is found, Inserted is set to False and Position designates the element with the matching key. Otherwise, Insert allocates a new node, initializes it to Key and New_Item, and adds it to Container; Inserted is set to True and Position designates the newly-inserted node. Any exception raised during allocation is propagated and Container is not modified. 45/2

```
{AI12-0112-1} procedure Insert (Container : in out Map;                               46/5
                                Key       : in     Key_Type;
                                Position  :    out Cursor;
                                Inserted  :    out Boolean)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                     or else raise Program_Error) and then
                 (Length (Container) <= Count_Type'Last - 1
                     or else raise Constraint_Error),
        Post => (declare
                    Original_Length : constant Count_Type :=
                       Length (Container)'Old;
                 begin
                   Has_Element (Container, Position) and then
                   (if Inserted then
                      Length (Container) = Original_Length + 1
                    else
                      Length (Container) = Original_Length));
```

{*AI95-00302-03*} Insert inserts Key into Container as per the five-parameter Insert, with the difference that an element initialized by default (see 3.3.1) is inserted. 47/2

48/5
```
{AI12-0112-1} procedure Insert (Container : in out Map;
                    Key       : in     Key_Type;
                    New_Item  : in     Element_Type)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                     or else raise Program_Error) and then
                (Length (Container) <= Count_Type'Last - 1
                     or else raise Constraint_Error),
        Post => Length (Container) = Length (Container)'Old + 1;
```

49/2 {*AI95-00302-03*} Insert inserts Key and New_Item into Container as per the five-parameter Insert, with the difference that if a node with a key equivalent to Key is already in the map, then Constraint_Error is propagated.

49.a/2 **Ramification:** This is equivalent to:

49.b/2
```
declare
  Inserted : Boolean; C : Cursor;
begin
  Insert (Container, Key, New_Item, C, Inserted);
  if not Inserted then
     raise Constraint_Error;
  end if;
end;
```

49.c/2 but doesn't require the hassle of **out** parameters.

50/5
```
{AI12-0112-1} procedure Include (Container : in out Map;
                    Key       : in     Key_Type;
                    New_Item  : in     Element_Type)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                     or else raise Program_Error) and then
                (Length (Container) <= Count_Type'Last - 1
                     or else raise Constraint_Error),
        Post => (declare
                   Original_Length : constant Count_Type :=
                      Length (Container)'Old;
                 begin
                   Length (Container)
                      in Original_Length | Original_Length + 1);
```

51/2 {*AI95-00302-03*} Include inserts Key and New_Item into Container as per the five-parameter Insert, with the difference that if a node with a key equivalent to Key is already in the map, then this operation assigns Key and New_Item to the matching node. Any exception raised during assignment is propagated.

51.a/2 **Ramification:** This is equivalent to:

51.b/2
```
declare
  C : Cursor := Find (Container, Key);
begin
  if C = No_Element then
     Insert (Container, Key, New_Item);
  else
     Replace (Container, Key, New_Item);
  end if;
end;
```

51.c/2 but this avoids doing the search twice.

52/5
```
{AI12-0112-1} procedure Replace (Container : in out Map;
                    Key       : in     Key_Type;
                    New_Item  : in     Element_Type)
   with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                     or else raise Program_Error,
        Post => Length (Container) = Length (Container)'Old;
```

53/2 {*AI95-00302-03*} Replace checks if a node with a key equivalent to Key is present in Container. If a match is found, Replace assigns Key and New_Item to the matching node; otherwise, Constraint_Error is propagated.

Discussion: We update the key as well as the element, as the key might include additional information that does not participate in equivalence. If only the element needs to be updated, use Replace_Element (Find (Container, Key), New_Element). 53.a/2

```
{AI12-0112-1} procedure Exclude (Container : in out Map;                    54/5
                  Key       : in     Key_Type)
   with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error,
        Post => (declare
                    Original_Length : constant Count_Type :=
                       Length (Container)'Old;
                 begin
                    Length (Container)
                       in Original_Length - 1 | Original_Length);
```

{*AI95-00302-03*} Exclude checks if a node with a key equivalent to Key is present in Container. If a match is found, Exclude removes the node from the map. 55/2

Ramification: Exclude should work on an empty map; nothing happens in that case. 55.a/2

```
{AI12-0112-1} procedure Delete (Container : in out Map;                     56/5
                  Key       : in     Key_Type)
   with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error,
        Post => Length (Container) = Length (Container)'Old - 1;
```

{*AI95-00302-03*} Delete checks if a node with a key equivalent to Key is present in Container. If a match is found, Delete removes the node from the map; otherwise, Constraint_Error is propagated. 57/2

```
procedure Delete (Container : in out Map;                                   58/5
                  Position  : in out Cursor)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error) and then
                (Position /= No_Element
                    or else raise Constraint_Error) and then
                (Has_Element (Container, Position)
                   or else raise Program_Error),
        Post => Length (Container) = Length (Container)'Old - 1 and then
                Position = No_Element;
```

{*AI95-00302-03*} {*AI12-0112-1*} Delete removes the node designated by Position from the map. 59/5

Ramification: The check on Position checks that the cursor does not belong to some other map. This check implies that a reference to the map is included in the cursor value. This wording is not meant to require detection of dangling cursors; such cursors are defined to be invalid, which means that execution is erroneous, and any result is allowed (including not raising an exception). 59.a/2

```
{AI12-0112-1} function First (Container : Map) return Cursor                60/5
   with Nonblocking, Global => null, Use_Formal => null,
        Post => (if not Is_Empty (Container)
                 then Has_Element (Container, First'Result)
                 else First'Result = No_Element);
```

{*AI95-00302-03*} If Length (Container) = 0, then First returns No_Element. Otherwise, First returns a cursor that designates the first node in Container. 61/2

```
{AI12-0112-1} function Next (Position : Cursor) return Cursor               62/5
   with Nonblocking, Global => in all, Use_Formal => null,
        Post => (if Position = No_Element then Next'Result = No_Element);
```

{*AI95-00302-03*} Returns a cursor that designates the successor of the node designated by Position. If Position designates the last node, then No_Element is returned. If Position equals No_Element, then No_Element is returned. 63/2

63.1/5
```
function Next (Container : Map;
               Position  : Cursor) return Cursor
   with Nonblocking, Global => null, Use_Formal => null,
        Pre  => Position = No_Element or else
                Has_Element (Container, Position)
                   or else raise Program_Error,
        Post => (if Position = No_Element then Next'Result = No_Element
                 elsif Next'Result = No_Element then
                   Position = Last (Container)
                 else Has_Element (Container, Next'Result));
```

63.2/5 {*AI12-0112-1*} Returns a cursor designating the successor of the node designated by Position in Container.

64/5
```
{AI12-0112-1} procedure Next (Position : in out Cursor)
   with Nonblocking, Global => in all, Use_Formal => null;
```

65/2 {*AI95-00302-03*} Equivalent to Position := Next (Position).

65.1/5
```
procedure Next (Container : in     Map;
                Position  : in out Cursor)
   with Nonblocking, Global => null, Use_Formal => null,
        Pre  => Position = No_Element or else
                Has_Element (Container, Position)
                   or else raise Program_Error,
        Post => (if Position /= No_Element
                 then Has_Element (Container, Position));
```

65.2/5 {*AI12-0112-1*} Equivalent to Position := Next (Container, Position).

66/5
```
{AI12-0112-1} function Find (Container : Map;
                             Key       : Key_Type) return Cursor
      with Post => (if Find'Result = No_Element
                    then Has_Element (Container, Find'Result));
```

67/2 {*AI95-00302-03*} If Length (Container) equals 0, then Find returns No_Element. Otherwise, Find checks if a node with a key equivalent to Key is present in Container. If a match is found, a cursor designating the matching node is returned; otherwise, No_Element is returned.

68/2
```
function Element (Container : Map;
                  Key       : Key_Type) return Element_Type;
```

69/2 {*AI95-00302-03*} Equivalent to Element (Find (Container, Key)).

70/2
```
function Contains (Container : Map;
                   Key       : Key_Type) return Boolean;
```

71/2 {*AI95-00302-03*} Equivalent to Find (Container, Key) /= No_Element.

Paragraphs 72 and 73 were moved above.

74/5
```
{AI12-0112-1} procedure Iterate
  (Container : in Map;
   Process   : not null access procedure (Position : in Cursor))
   with Allows_Exit;
```

75/3 {*AI95-00302-03*} {*AI05-0265-1*} Iterate calls Process.**all** with a cursor that designates each node in Container, starting with the first node and moving the cursor according to the successor relation. Tampering with the cursors of Container is prohibited during the execution of a call on Process.**all**. Any exception raised by Process.**all** is propagated.

75.a/2 **Implementation Note:** The “tamper with cursors” check takes place when the operations that insert or delete elements, and so on, are called.

75.b/2 See Iterate for vectors (A.18.2) for a suggested implementation of the check.

{*AI12-0111-1*} The nested package Stable provides a type Stable.Map that represents a *stable* map, which is one that cannot grow and shrink. Such a map can be created by calling the Copy function, or by establishing a *stabilized view* of an ordinary map. 75.1/5

{*AI12-0111-1*} The subprograms of the map package that have a parameter or result of type Map are included in the nested package Stable with the same specification, except that the following are omitted: 75.2/5

Tampering_With_Cursors_Prohibited, Tampering_With_Elements_Prohibited, Assign, Move, Insert, Include, Clear, Delete, Exclude, (for Ordered_Maps) Delete_First and Delete_Last, and (for Hashed_Maps) Reserve_Capacity 75.3/5

Ramification: The names Map and Cursor mean the types declared in the nested package in these subprogram specifications. 75.b.1/5

Reason: The omitted routines are those that tamper with cursors or elements (or test that state). The model is that it is impossible to tamper with cursors or elements of a stable view since no such operations are included. Thus tampering checks are not needed for a stable view, and we omit the operations associated with those checks. 75.b.2/5

{*AI12-0111-1*} The operations of this package are equivalent to those for ordinary maps, except that the calls to Tampering_With_Cursors_Prohibited and Tampering_With_Elements_Prohibited that occur in preconditions are replaced by False, and any that occur in postconditions are replaced by True. 75.4/5

{*AI12-0111-1*} {*AI12-0439-1*} If a stable map is declared with the Base discriminant designating a pre-existing ordinary map, the stable map represents a stabilized view of the underlying ordinary map, and any operation on the stable map is reflected on the underlying ordinary map. While a stabilized view exists, any operation that tampers with elements performed on the underlying map is prohibited. The finalization of a stable map that provides such a view removes this restriction on the underlying ordinary map [(though some other restriction can exist due to other concurrent iterations or stabilized views)]. 75.5/5

{*AI12-0111-1*} {*AI12-0438-1*} If a stable map is declared without specifying Base, the object is necessarily initialized. The initializing expression of the stable map, [typically a call on Copy], determines the Length of the map. The Length of a stable map never changes after initialization. 75.6/5

Proof: {*AI12-0438-1*} Initialization is required as the type is indefinite, see 3.3.1. 75.c/5

Bounded (Run-Time) Errors

{*AI05-0022-1*} {*AI05-0248-1*} It is a bounded error for the actual function associated with a generic formal subprogram, when called as part of an operation of a map package, to tamper with elements of any map parameter of the operation. Either Program_Error is raised, or the operation works as defined on the value of the map either prior to, or subsequent to, some or all of the modifications to the map. 75.7/3

{*AI05-0027-1*} It is a bounded error to call any subprogram declared in the visible part of a map package when the associated container has been finalized. If the operation takes Container as an **in out** parameter, then it raises Constraint_Error or Program_Error. Otherwise, the operation either proceeds as it would for an empty container, or it raises Constraint_Error or Program_Error. 75.8/3

Erroneous Execution

{*AI95-00302-03*} A Cursor value is *invalid* if any of the following have occurred since it was created: 76/2

- The map that contains the node it designates has been finalized; 77/2
- {*AI05-0160-1*} The map that contains the node it designates has been used as the Target of a call to Assign, or as the target of an assignment_statement; 77.1/3
- The map that contains the node it designates has been used as the Source or Target of a call to Move; or 78/2
- {*AI05-0160-1*} {*AI05-0262-1*} The node it designates has been removed from the map that previously contained the node. 79/3

79.a/3 **Ramification:** {*AI05-0160-1*} This can happen directly via calls to Clear, Exclude, and Delete.

80/2 The result of "=" or Has_Element is unspecified if these functions are called with an invalid cursor parameter. Execution is erroneous if any other subprogram declared in Containers.Hashed_Maps or Containers.Ordered_Maps is called with an invalid cursor parameter.

80.a/2 **Discussion:** The list above is intended to be exhaustive. In other cases, a cursor value continues to designate its original element. For instance, cursor values survive the insertion and deletion of other nodes.

80.b/2 While it is possible to check for these cases, in many cases the overhead necessary to make the check is substantial in time or space. Implementations are encouraged to check for as many of these cases as possible and raise Program_Error if detected.

80.1/3 {*AI05-0212-1*} Execution is erroneous if the map associated with the result of a call to Reference or Constant_Reference is finalized before the result object returned by the call to Reference or Constant_Reference is finalized.

80.c/3 **Reason:** Each object of Reference_Type and Constant_Reference_Type probably contains some reference to the originating container. If that container is prematurely finalized (which is only possible via Unchecked_Deallocation, as accessibility checks prevent passing a container to Reference that will not live as long as the result), the finalization of the object of Reference_Type will try to access a nonexistent object. This is a normal case of a dangling pointer created by Unchecked_Deallocation; we have to explicitly mention it here as the pointer in question is not visible in the specification of the type. (This is the same reason we have to say this for invalid cursors.)

Implementation Requirements

81/5 {*AI95-00302-03*} {*AI12-0437-1*} No storage associated with a map object shall be lost upon assignment or scope exit.

82/3 {*AI95-00302-03*} {*AI05-0262-1*} The execution of an assignment_statement for a map shall have the effect of copying the elements from the source map object to the target map object and changing the length of the target object to that of the source object.

82.a/3 **Implementation Note:** {*AI05-0298-1*} An assignment of a Map is a “deep” copy; that is the elements are copied as well as the data structures. We say “effect of” in order to allow the implementation to avoid copying elements immediately if it wishes. For instance, an implementation that avoided copying until one of the containers is modified would be allowed. (Note that this implementation would require care, see A.18.2 for more.)

Implementation Advice

83/2 {*AI95-00302-03*} Move should not copy elements, and should minimize copying of internal data structures.

83.a/2 **Implementation Advice:** Move for a map should not copy elements, and should minimize copying of internal data structures.

83.b/2 **Implementation Note:** Usually that can be accomplished simply by moving the pointer(s) to the internal data structures from the Source container to the Target container.

84/2 {*AI95-00302-03*} If an exception is propagated from a map operation, no storage should be lost, nor any elements removed from a map unless specified by the operation.

84.a/2 **Implementation Advice:** If an exception is propagated from a map operation, no storage should be lost, nor any elements removed from a map unless specified by the operation.

84.b/2 **Reason:** This is important so that programs can recover from errors. But we don't want to require heroic efforts, so we just require documentation of cases where this can't be accomplished.

Wording Changes from Ada 95

84.c/2 {*AI95-00302-03*} This description of maps is new; the extensions are documented with the specific packages.

Extensions to Ada 2005

84.d/3 {*AI05-0212-1*} Added reference support to make map containers more convenient to use.

Wording Changes from Ada 2005

{*AI05-0001-1*} Added procedure Assign; the extension and incompatibility is documented with the specific packages. 84.e/3

{*AI05-0001-1*} Generalized the definition of Move. Specified which elements are read/written by stream attributes. 84.f/3

{*AI05-0022-1*} **Correction:** Added a Bounded (Run-Time) Error to cover tampering by generic actual subprograms. 84.g/3

{*AI05-0027-1*} **Correction:** Added a Bounded (Run-Time) Error to cover access to finalized map containers. 84.h/3

{*AI05-0160-1*} **Correction:** Revised the definition of invalid cursors to cover missing (and new) cases. 84.i/3

{*AI05-0265-1*} **Correction:** Defined when a container prohibits tampering in order to more clearly define where the check is made and the exception raised. 84.j/3

Inconsistencies With Ada 2012

{*AI12-0111-1*} **Correction:** Tampering with elements is now defined to be equivalent to tampering with cursors for ordinary containers. If a program requires tampering detection to work, it might fail in Ada 2022. Needless to say, this shouldn't happen outside of test programs. See Inconsistencies With Ada 2012 in A.18.2 for more details. 84.k/5

Extensions to Ada 2012

{*AI12-0196-1*} **Correction:** Replace_Element is now defined such that it can be used concurrently so long as it operates on different elements. This allows some container operations to be used in parallel without separate synchronization. 84.l/5

Wording Changes from Ada 2012

{*AI12-0110-1*} **Correction:** Clarified that tampering checks precede all other checks made by a subprogram (but come after those associated with the call). 84.m/4

{*AI12-0112-1*} Added contracts to this container. This includes describing some of the semantics with pre- and postconditions, rather than English text. Note that the preconditions can be Suppressed (see 11.5). 84.n/5

A.18.5 The Generic Package Containers.Hashed_Maps

Static Semantics

{*AI95-00302-03*} The generic library package Containers.Hashed_Maps has the following declaration: 1/2

```
{AI05-0084-1} {AI05-0212-1} {AI12-0112-1} with Ada.Iterator_Interfaces;                2/5
generic
   type Key_Type is private;
   type Element_Type is private;
   with function Hash (Key : Key_Type) return Hash_Type;
   with function Equivalent_Keys (Left, Right : Key_Type)
      return Boolean;
   with function "=" (Left, Right : Element_Type)
      return Boolean is <>;
package Ada.Containers.Hashed_Maps
   with Preelaborate, Remote_Types,
        Nonblocking, Global => in out synchronized is
```

Discussion: {*AI12-0112-1*} For discussion on the reasons and meaning of the specifications of the Global and Nonblocking aspects in this generic package, see the notes on the equivalent operations in the specification of the Containers.Vectors package (see A.18.2). 2.a/5

3/5
```
{AI05-0212-1} {AI12-0111-1} {AI12-0112-1} {AI12-0212-1} {AI12-0339-1} {AI12-0339-1}
type Map is tagged private
      with Constant_Indexing => Constant_Reference,
           Variable_Indexing => Reference,
           Default_Iterator  => Iterate,
           Iterator_Element  => Element_Type,
           Iterator_View     => Stable.Map,
           Aggregate         => (Empty     => Empty,
                                 Add_Named => Insert),
           Stable_Properties => (Length,
                                 Tampering_With_Cursors_Prohibited,
                                 Tampering_With_Elements_Prohibited),
           Default_Initial_Condition =>
              Length (Map) = 0 and then
              (not Tampering_With_Cursors_Prohibited (Map)) and then
              (not Tampering_With_Elements_Prohibited (Map)),
           Preelaborable_Initialization;
```

3.a/5 **Discussion:** {*AI12-0112-1*} Unlike a Vector, the Stable_Properties of a Hashed_Map do not include the Capacity. If we had included it, some of the otherwise shared definitions would need different postconditions for Hashed_Maps and Ordered_Maps. If we were starting these containers from scratch, we probably would have approached the sharing of definitions differently so that we could avoid issues like this, but a major reorganization of this existing material would be too much change.

4/5
```
{AI12-0399-1}    type Cursor is private
      with Preelaborable_Initialization;
```

5/2
```
   Empty_Map : constant Map;
```

6/2
```
   No_Element : constant Cursor;
```

6.1/5
```
{AI05-0212-1} {AI12-0112-1}    function Has_Element (Position : Cursor) return
Boolean
      with Nonblocking, Global => in all, Use_Formal => null;
```

6.2/5
```
{AI12-0112-1}    function Has_Element (Container : Map; Position : Cursor)
      return Boolean
      with Nonblocking, Global => null, Use_Formal => null;
```

6.3/5
```
{AI05-0212-1}    package Map_Iterator_Interfaces is new
       Ada.Iterator_Interfaces (Cursor, Has_Element);
```

7/2
```
   function "=" (Left, Right : Map) return Boolean;
```

7.1/5
```
{AI12-0112-1}    function Tampering_With_Cursors_Prohibited
      (Container : Map) return Boolean
      with Nonblocking, Global => null, Use_Formal => null;
```

7.2/5
```
{AI12-0112-1}    function Tampering_With_Elements_Prohibited
      (Container : Map) return Boolean
      with Nonblocking, Global => null, Use_Formal => null;
```

7.3/5
```
{AI12-0339-1}    function Empty (Capacity : Count_Type := implementation-defined)
      return Map
      with Post =>
              Capacity (Empty'Result) >= Capacity and then
              not Tampering_With_Elements_Prohibited (Empty'Result) and then
              not Tampering_With_Cursors_Prohibited (Empty'Result) and then
              Length (Empty'Result) = 0;
```

8/5
```
{AI12-0112-1}    function Capacity (Container : Map) return Count_Type
      with Nonblocking, Global => null, Use_Formal => null;
```

9/5
```
{AI12-0112-1}    procedure Reserve_Capacity (Container : in out Map;
                                             Capacity  : in     Count_Type)
      with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                   or else raise Program_Error,
           Post => Container.Capacity >= Capacity;
```

10/5
```
{AI12-0112-1}    function Length (Container : Map) return Count_Type
      with Nonblocking, Global => null, Use_Formal => null;
```

11/5
```
{AI12-0112-1}    function Is_Empty (Container : Map) return Boolean
      with Nonblocking, Global => null, Use_Formal => null,
           Post => Is_Empty'Result = (Length (Container) = 0);
```

```
{AI12-0112-1}    procedure Clear (Container : in out Map)                                    12/5
      with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                       or else raise Program_Error,
           Post => Capacity (Container) = Capacity (Container)'Old and then
                   Length (Container) = 0;

{AI12-0112-1}    function Key (Position : Cursor) return Key_Type                            13/5
      with Pre  => Position /= No_Element
                       or else raise Constraint_Error,
           Nonblocking, Global => in all, Use_Formal => Key_Type;

{AI12-0112-1}    function Key (Container : Map;                                              13.1/5
                  Position : Cursor) return Key_Type
      with Pre  => (Position /= No_Element
                       or else raise Constraint_Error) and then
                   (Has_Element (Container, Position)
                       or else raise Program_Error),
           Nonblocking, Global => null, Use_Formal => Key_Type;

{AI12-0112-1}    function Element (Position : Cursor) return Element_Type                    14/5
      with Pre  => Position /= No_Element
                       or else raise Constraint_Error,
           Nonblocking, Global => in all, Use_Formal => Element_Type;

{AI12-0112-1}    function Element (Container : Map;                                          14.1/5
                       Position  : Cursor) return Element_Type
      with Pre  => (Position /= No_Element
                       or else raise Constraint_Error) and then
                   (Has_Element (Container, Position)
                     or else raise Program_Error),
           Nonblocking, Global => null, Use_Formal => Element_Type;

{AI12-0112-1}    procedure Replace_Element (Container : in out Map;                          15/5
                                Position  : in     Cursor;
                                New_item  : in     Element_Type)
      with Pre  => (not Tampering_With_Elements_Prohibited (Container)
                       or else raise Program_Error) and then
                   (Position /= No_Element
                       or else raise Constraint_Error) and then
                   (Has_Element (Container, Position)
                       or else raise Program_Error);

{AI12-0112-1}    procedure Query_Element                                                     16/5
     (Position : in Cursor;
      Process  : not null access procedure (Key     : in Key_Type;
                                            Element : in Element_Type))
      with Pre  => Position /= No_Element
                       or else raise Constraint_Error,
           Global => in all;

{AI12-0112-1}    procedure Query_Element                                                     16.1/5
     (Container : in Map;
      Position  : in Cursor;
      Process   : not null access procedure (Key     : in Key_Type;
                                             Element : in Element_Type))
      with Pre  => (Position /= No_Element
                       or else raise Constraint_Error) and then
                   (Has_Element (Container, Position)
                       or else raise Program_Error);

{AI12-0112-1}    procedure Update_Element                                                    17/5
     (Container : in out Map;
      Position  : in     Cursor;
      Process   : not null access procedure
                        (Key     : in     Key_Type;
                         Element : in out Element_Type))
      with Pre  => (Position /= No_Element
                       or else raise Constraint_Error) and then
                   (Has_Element (Container, Position)
                       or else raise Program_Error);
```

17.1/5 {*AI05-0212-1*} {*AI12-0112-1*}
```
type Constant_Reference_Type
   (Element : not null access constant Element_Type) is private
with Implicit_Dereference => Element,
     Nonblocking, Global => in out synchronized,
     Default_Initial_Condition => (raise Program_Error);
```

17.2/5 {*AI05-0212-1*} {*AI12-0112-1*}
```
type Reference_Type (Element : not null access
Element_Type) is private
with Implicit_Dereference => Element,
     Nonblocking, Global => in out synchronized,
     Default_Initial_Condition => (raise Program_Error);
```

17.3/5 {*AI05-0212-1*} {*AI12-0112-1*}
```
function Constant_Reference (Container : aliased in
Map;
                             Position  : in Cursor)
return Constant_Reference_Type
with Pre  => (Position /= No_Element
                 or else raise Constraint_Error) and then
              (Has_Element (Container, Position)
                 or else raise Program_Error),
     Post => Tampering_With_Cursors_Prohibited (Container),
     Nonblocking, Global => null, Use_Formal => null;
```

17.4/5 {*AI05-0212-1*} {*AI12-0112-1*}
```
function Reference (Container : aliased in out Map;
                    Position  : in Cursor)
return Reference_Type
with Pre  => (Position /= No_Element
                 or else raise Constraint_Error) and then
              (Has_Element (Container, Position)
                 or else raise Program_Error),
     Post => Tampering_With_Cursors_Prohibited (Container),
     Nonblocking, Global => null, Use_Formal => null;
```

17.5/5 {*AI05-0212-1*} {*AI12-0112-1*}
```
function Constant_Reference (Container : aliased in
Map;
                             Key       : in Key_Type)
return Constant_Reference_Type
with Pre  => Find (Container, Key) /= No_Element
                 or else raise Constraint_Error,
     Post => Tampering_With_Cursors_Prohibited (Container),
     Nonblocking, Global => null, Use_Formal => null;
```

17.6/5 {*AI05-0212-1*} {*AI12-0112-1*}
```
function Reference (Container : aliased in out Map;
                    Key       : in Key_Type)
return Reference_Type
with Pre  => Find (Container, Key) /= No_Element
                 or else raise Constraint_Error,
     Post => Tampering_With_Cursors_Prohibited (Container),
     Nonblocking, Global => null, Use_Formal => null;
```

17.7/5 {*AI05-0001-1*} {*AI12-0112-1*}
```
procedure Assign (Target : in out Map; Source : in
Map)
with Pre  => not Tampering_With_Cursors_Prohibited (Target)
                 or else raise Program_Error,
     Post => Length (Source) = Length (Target) and then
             Capacity (Target) >= Length (Source);
```

17.8/5 {*AI05-0001-1*} {*AI12-0112-1*}
```
function Copy (Source : Map; Capacity : Count_Type
:= 0)
return Map
with Pre  => Capacity = 0 or else Capacity >= Length (Source)
              or else raise Capacity_Error,
     Post =>
        Length (Copy'Result) = Length (Source) and then
        not Tampering_With_Elements_Prohibited (Copy'Result) and then
        not Tampering_With_Cursors_Prohibited (Copy'Result) and then
        Copy'Result.Capacity = (if Capacity = 0 then
           Length (Source) else Capacity);
```

```
{AI12-0112-1}    procedure Move (Target : in out Map;                                    18/5
                   Source : in out Map)
      with Pre  => (not Tampering_With_Cursors_Prohibited (Target)
                      or else raise Program_Error) and then
                   (not Tampering_With_Cursors_Prohibited (Source)
                      or else raise Program_Error),
           Post => (if not Target'Has_Same_Storage (Source) then
                      Length (Target) = Length (Source'Old) and then
                      Length (Source) = 0);

{AI12-0112-1}    procedure Insert (Container : in out Map;                               19/5
                       Key       : in     Key_Type;
                       New_Item  : in     Element_Type;
                       Position  :    out Cursor;
                       Inserted  :    out Boolean)
      with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                      or else raise Program_Error) and then
                   (Length (Container) <= Count_Type'Last - 1
                      or else raise Constraint_Error),
           Post => (declare
                      Original_Length : constant Count_Type :=
                         Length (Container)'Old;
                    begin
                      Has_Element (Container, Position) and then
                     (if Inserted then
                        Length (Container) = Original_Length + 1
                      else
                        Length (Container) = Original_Length)) and then
                    Capacity (Container) >= Length (Container);

{AI12-0112-1}    procedure Insert (Container : in out Map;                               20/5
                       Key       : in     Key_Type;
                       Position  :    out Cursor;
                       Inserted  :    out Boolean)
      with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                       or else raise Program_Error) and then
                   (Length (Container) <= Count_Type'Last - 1
                       or else raise Constraint_Error),
           Post => (declare
                      Original_Length : constant Count_Type :=
                         Length (Container)'Old;
                    begin
                      Has_Element (Container, Position) and then
                     (if Inserted then
                        Length (Container) = Original_Length + 1
                      else
                        Length (Container) = Original_Length)) and then
                    Capacity (Container) >= Length (Container);

{AI12-0112-1}    procedure Insert (Container : in out Map;                               21/5
                       Key       : in     Key_Type;
                       New_Item  : in     Element_Type)
      with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                       or else raise Program_Error) and then
                   (Length (Container) <= Count_Type'Last - 1
                       or else raise Constraint_Error),
           Post => Length (Container) = Length (Container)'Old + 1 and then
                   Capacity (Container) >= Length (Container);
```

```
22/5    {AI12-0112-1}    procedure Include (Container : in out Map;
                                           Key       : in     Key_Type;
                                           New_Item  : in     Element_Type)
                         with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                                          or else raise Program_Error) and then
                                      (Length (Container) <= Count_Type'Last - 1
                                          or else raise Constraint_Error),
                              Post => (declare
                                         Original_Length : constant Count_Type :=
                                           Length (Container)'Old;
                                       begin
                                         Length (Container)
                                           in Original_Length | Original_Length + 1) and then
                                       Capacity (Container) >= Length (Container);

23/5    {AI12-0112-1}    procedure Replace (Container : in out Map;
                                           Key       : in     Key_Type;
                                           New_Item  : in     Element_Type)
                         with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                                          or else raise Program_Error,
                              Post => Length (Container) = Length (Container)'Old;

24/5    {AI12-0112-1}    procedure Exclude (Container : in out Map;
                                           Key       : in     Key_Type)
                         with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                                          or else raise Program_Error,
                              Post => (declare
                                         Original_Length : constant Count_Type :=
                                           Length (Container)'Old;
                                       begin
                                         Length (Container)
                                           in Original_Length - 1 | Original_Length);

25/5    {AI12-0112-1}    procedure Delete (Container : in out Map;
                                           Key       : in     Key_Type)
                         with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                                          or else raise Program_Error,
                              Post => Length (Container) = Length (Container)'Old - 1;

26/5    {AI12-0112-1}    procedure Delete (Container : in out Map;
                                           Position  : in out Cursor)
                         with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                                          or else raise Program_Error) and then
                                      (Position /= No_Element
                                          or else raise Constraint_Error) and then
                                      (Has_Element (Container, Position)
                                          or else raise Program_Error),
                              Post => Length (Container) = Length (Container)'Old - 1 and then
                                      Position = No_Element;

27/5    {AI12-0112-1}    function First (Container : Map) return Cursor
                         with Nonblocking, Global => null, Use_Formal => null,
                              Post => (if not Is_Empty (Container)
                                       then Has_Element (Container, First'Result)
                                       else First'Result = No_Element);

28/5    {AI12-0112-1}    function Next (Position : Cursor) return Cursor
                         with Nonblocking, Global => in all, Use_Formal => null,
                              Post => (if Position = No_Element then Next'Result = No_Element);

28.1/5  {AI12-0112-1}    function Next (Container : Map;
                                        Position : Cursor) return Cursor
                         with Nonblocking, Global => null, Use_Formal => null,
                              Pre  => Position = No_Element or else
                                      Has_Element (Container, Position)
                                         or else raise Program_Error,
                              Post => (if Position = No_Element then Next'Result = No_Element
                                       elsif Next'Result = No_Element then
                                         Position = Last (Container)
                                       else Has_Element (Container, Next'Result));
```

```
{AI12-0112-1}    procedure Next (Position : in out Cursor)                                    29/5
      with Nonblocking, Global => in all, Use_Formal => null;

{AI12-0112-1}    procedure Next (Container : in     Map;                                     29.1/5
                   Position  : in out Cursor)
      with Nonblocking, Global => null, Use_Formal => null,
           Pre  => Position = No_Element or else
                   Has_Element (Container, Position)
                      or else raise Program_Error,
           Post => (if Position /= No_Element
                    then Has_Element (Container, Position));

{AI12-0112-1}    function Find (Container : Map;                                              30/5
                  Key       : Key_Type)
      return Cursor
      with Post => (if Find'Result /= No_Element
                    then Has_Element (Container, Find'Result));

   function Element (Container : Map;                                                        31/2
                     Key       : Key_Type)
      return Element_Type;

   function Contains (Container : Map;                                                       32/2
                      Key       : Key_Type) return Boolean;

This paragraph was deleted.{AI05-0212-1}                                                     33/3

{AI12-0112-1}    function Equivalent_Keys (Left, Right : Cursor)                             34/5
      return Boolean
      with Pre    => (Left /= No_Element and then Right /= No_Element)
                        or else raise Constraint_Error,
           Global => in all;

{AI12-0112-1}    function Equivalent_Keys (Left  : Cursor;                                   35/5
                           Right : Key_Type)
      return Boolean
      with Pre    => Left /= No_Element or else raise Constraint_Error,
           Global => in all;

{AI12-0112-1}    function Equivalent_Keys (Left  : Key_Type;                                 36/5
                           Right : Cursor)
      return Boolean
      with Pre    => Right /= No_Element or else raise Constraint_Error,
           Global => in all;

{AI12-0112-1}    procedure Iterate                                                           37/5
     (Container : in Map;
      Process   : not null access procedure (Position : in Cursor))
      with Allows_Exit;

{AI05-0212-1} {AI12-0112-1} {AI12-0266-1}    function Iterate (Container : in Map)          37.1/5
      return Map_Iterator_Interfaces.Parallel_Iterator'Class
      with Post => Tampering_With_Cursors_Prohibited (Container);

{AI12-0111-1}    package Stable is                                                           37.2/5

{AI12-0111-1} {AI12-0339-1} {AI12-0399-1} {AI12-0407-1}       type Map (Base : not       37.3/5
null access Hashed_Maps.Map) is
         tagged limited private
         with Constant_Indexing => Constant_Reference,
              Variable_Indexing => Reference,
              Default_Iterator  => Iterate,
              Iterator_Element  => Element_Type,
              Stable_Properties => (Length),
              Global            => null,
              Default_Initial_Condition => Length (Map) = 0,
              Preelaborable_Initialization;

{AI12-0111-1} {AI12-0399-1}       type Cursor is private                                    37.4/5
      with Preelaborable_Initialization;

{AI12-0111-1}       Empty_Map : constant Map;                                              37.5/5

{AI12-0111-1}       No_Element : constant Cursor;                                          37.6/5
```

```
37.7/5      {AI12-0111-1}       function Has_Element (Position : Cursor) return Boolean
                with Nonblocking, Global => in all, Use_Formal => null;
37.8/5      {AI12-0111-1}       package Map_Iterator_Interfaces is new
                Ada.Iterator_Interfaces (Cursor, Has_Element);
37.9/5      {AI12-0111-1}       procedure Assign (Target : in out Hashed_Maps.Map;
                                    Source : in Map)
                with Post => Length (Source) = Length (Target);
37.10/5     {AI12-0111-1}       function Copy (Source : Hashed_Maps.Map) return Map
                with Post => Length (Copy'Result) = Length (Source);
37.11/5     {AI12-0111-1}       type Constant_Reference_Type
                    (Element : not null access constant Element_Type) is private
                with Implicit_Dereference => Element,
                     Nonblocking, Global => null, Use_Formal => null,
                     Default_Initial_Condition => (raise Program_Error);
37.12/5     {AI12-0111-1}       type Reference_Type
                    (Element : not null access Element_Type) is private
                with Implicit_Dereference => Element,
                     Nonblocking, Global => null, Use_Formal => null,
                     Default_Initial_Condition => (raise Program_Error);
37.13/5     {AI12-0111-1}       -- Additional subprograms as described in the text
            -- are declared here.
37.14/5     {AI12-0111-1}   private
37.15/5     {AI12-0111-1}       ... -- not specified by the language
37.16/5     {AI12-0111-1}   end Stable;
38/2        private
39/2            ... -- not specified by the language
40/2        end Ada.Containers.Hashed_Maps;
```

41/2 {*AI95-00302-03*} An object of type Map contains an expandable hash table, which is used to provide direct access to nodes. The *capacity* of an object of type Map is the maximum number of nodes that can be inserted into the hash table prior to it being automatically expanded.

41.a/2 **Implementation Note:** The expected implementation for a Map uses a hash table which is grown when it is too small, with linked lists hanging off of each bucket. Note that in that implementation a cursor needs a back pointer to the Map object to implement iteration; that could either be in the nodes, or in the cursor object. To provide an average *O*(1) access time, capacity would typically equal the number of buckets in such an implementation, so that the average bucket linked list length would be no more than 1.0.

41.b/2 There is no defined relationship between elements in a hashed map. Typically, iteration will return elements in the order that they are hashed in.

42/2 {*AI95-00302-03*} Two keys *K1* and *K2* are defined to be *equivalent* if Equivalent_Keys (*K1*, *K2*) returns True.

43/2 {*AI95-00302-03*} The actual function for the generic formal function Hash is expected to return the same value each time it is called with a particular key value. For any two equivalent key values, the actual for Hash is expected to return the same value. If the actual for Hash behaves in some other manner, the behavior of this package is unspecified. Which subprograms of this package call Hash, and how many times they call it, is unspecified.

43.a/2 **Implementation Note:** The implementation is not required to protect against Hash raising an exception, or returning random numbers, or any other “bad” behavior. It's not practical to do so, and a broken Hash function makes the container unusable.

43.b/2 The implementation can call Hash whenever it is needed; we don't want to specify how often that happens. The result must remain the same (this is logically a pure function), or the behavior is unspecified.

44/2 {*AI95-00302-03*} The actual function for the generic formal function Equivalent_Keys on Key_Type values is expected to return the same value each time it is called with a particular pair of key values. It should define an equivalence relationship, that is, be reflexive, symmetric, and transitive. If the actual for

Equivalent_Keys behaves in some other manner, the behavior of this package is unspecified. Which subprograms of this package call Equivalent_Keys, and how many times they call it, is unspecified.

Implementation Note: As with Hash, the implementation is not required to protect against Equivalent_Keys raising an exception or returning random results. Similarly, the implementation can call this operation whenever it is needed. The result must remain the same (this is a logically pure function), or the behavior is unspecified. 44.a/2

{*AI95-00302-03*} If the value of a key stored in a node of a map is changed other than by an operation in this package such that at least one of Hash or Equivalent_Keys give different results, the behavior of this package is unspecified. 45/2

Implementation Note: The implementation is not required to protect against changes to key values other than via the operations declared in the Hashed_Maps package. 45.a/2

To see how this could happen, imagine an instance of Hashed_Maps where the key type is an access-to-variable type and Hash returns a value derived from the components of the designated object. Then, any operation that has a key value could modify those components and change the hash value: 45.b/2

```
Key (Map).Some_Component := New_Value;
```
45.c/2

This is really a design error on the part of the user of the map; it shouldn't be possible to modify keys stored in a map. But we can't prevent this error anymore than we can prevent someone passing as Hash a random number generator. 45.d/2

{*AI95-00302-03*} Which nodes are the first node and the last node of a map, and which node is the successor of a given node, are unspecified, other than the general semantics described in A.18.4. 46/2

Implementation Note: Typically the first node will be the first node in the first bucket, the last node will be the last node in the last bucket, and the successor will be obtained by following the collision list, and going to the next bucket at the end of each bucket. 46.a/2

```
function Empty (Capacity : Count_Type := implementation-defined)
   return Map
   with Post =>
           Capacity (Empty'Result) >= Capacity and then
           not Tampering_With_Elements_Prohibited (Empty'Result) and then
           not Tampering_With_Cursors_Prohibited (Empty'Result) and then
           Length (Empty'Result) = 0;
```
46.1/5

{*AI12-0339-1*} Returns an empty map. 46.2/2

```
{AI12-0112-1} function Capacity (Container : Map) return Count_Type
   with Nonblocking, Global => null, Use_Formal => null;
```
47/5

{*AI95-00302-03*} Returns the capacity of Container. 48/2

```
{AI12-0112-1} procedure Reserve_Capacity (Container : in out Map;
                                          Capacity  : in     Count_Type)
   with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                or else raise Program_Error,
        Post => Container.Capacity >= Capacity;
```
49/5

{*AI95-00302-03*} Reserve_Capacity allocates a new hash table such that the length of the resulting map can become at least the value Capacity without requiring an additional call to Reserve_Capacity, and is large enough to hold the current length of Container. Reserve_Capacity then rehashes the nodes in Container onto the new hash table. It replaces the old hash table with the new hash table, and then deallocates the old hash table. Any exception raised during allocation is propagated and Container is not modified. 50/2

This paragraph was deleted.{*AI12-0112-1*} 51/5

Implementation Note: This routine is used to preallocate the internal hash table to the specified capacity such that future Inserts do not require expansion of the hash table. Therefore, the implementation should allocate the needed memory to make that true at this point, even though the visible semantics could be preserved by waiting until enough elements are inserted. 51.a/2

{*AI05-0005-1*} While Reserve_Capacity can be used to reduce the capacity of a map, we do not specify whether an implementation actually supports reduction of the capacity. Since the actual capacity can be anything greater than or equal to Capacity, an implementation never has to reduce the capacity. 51.b/3

51.c/2 Reserve_Capacity tampers with the cursors, as rehashing probably will change the order that elements are stored in the map.

52/5
```
{AI12-0112-1} procedure Clear (Container : in out Map)
   with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error,
        Post => Capacity (Container) = Capacity (Container)'Old and then
                Length (Container) = 0;
```

53/2 {*AI95-00302-03*} In addition to the semantics described in A.18.4, Clear does not affect the capacity of Container.

53.1/5
```
{AI12-0112-1} procedure Assign (Target : in out Map; Source : in Map)
      with Pre  => not Tampering_With_Cursors_Prohibited (Target)
                       or else raise Program_Error,
           Post => Length (Source) = Length (Target) and then
                   Capacity (Target) >= Length (Source);
```

53.2/3 {*AI05-0001-1*} {*AI05-0248-1*} In addition to the semantics described in A.18.4, if the length of Source is greater than the capacity of Target, Reserve_Capacity (Target, Length (Source)) is called before assigning any elements.

53.3/5
```
function Copy (Source : Map; Capacity : Count_Type := 0)
   return Map
   with Pre  => Capacity = 0 or else Capacity >= Length (Source)
                   or else raise Capacity_Error,
        Post =>
           Length (Copy'Result) = Length (Source) and then
           not Tampering_With_Elements_Prohibited (Copy'Result) and then
           not Tampering_With_Cursors_Prohibited (Copy'Result) and then
           Copy'Result.Capacity = (if Capacity = 0 then
              Length (Source) else Capacity);
```

53.4/5 {*AI05-0001-1*} {*AI12-0112-1*} Returns a map whose keys and elements are initialized from the keys and elements of Source.

53.a/2 **Implementation Note:** In:

53.b/2
```
procedure Move (Target : in out Map;
                Source : in out Map);
```

53.c/2 The intended implementation is that the internal hash table of Target is first deallocated; then the internal hash table is removed from Source and moved to Target.

54/5
```
{AI12-0112-1} procedure Insert (Container : in out Map;
                  Key       : in     Key_Type;
                  New_Item  : in     Element_Type;
                  Position  :    out Cursor;
                  Inserted  :    out Boolean)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                     or else raise Program_Error) and then
                (Length (Container) <= Count_Type'Last - 1
                     or else raise Constraint_Error),
        Post => (declare
                   Original_Length : constant Count_Type :=
                      Length (Container)'Old;
                 begin
                   Has_Element (Container, Position) and then
                  (if Inserted then
                     Length (Container) = Original_Length + 1
                   else
                     Length (Container) = Original_Length)) and then
                 Capacity (Container) >= Length (Container);
```

55/2 {*AI95-00302-03*} In addition to the semantics described in A.18.4, if Length (Container) equals Capacity (Container), then Insert first calls Reserve_Capacity to increase the capacity of Container to some larger value.

Implementation Note: Insert should only compare keys that hash to the same bucket in the hash table. 55.a/2

We specify when Reserve_Capacity is called to bound the overhead of capacity expansion operations (which are potentially expensive). Moreover, expansion can be predicted by comparing Capacity(Map) to Length(Map). Since we don't specify by how much the hash table is expanded, this only can be used to predict the next expansion, not later ones. 55.b/2

Implementation Note: In: 55.c/2

```
procedure Exclude (Container : in out Map;
                   Key       : in     Key_Type);
```
55.d/2

Exclude should only compare keys that hash to the same bucket in the hash table. 55.e/2

Implementation Note: In: 55.f/2

```
procedure Delete (Container : in out Map;
                  Key       : in     Key_Type);
```
55.g/2

Delete should only compare keys that hash to the same bucket in the hash table. The node containing the element may be deallocated now, or it may be saved and reused later. 55.h/2

Implementation Note: In: 55.i/2

```
function First (Container : Map) return Cursor;
```
55.j/2

In a typical implementation, this will be the first node in the lowest numbered hash bucket that contains a node. 55.k/2

Implementation Note: In: 55.l/2

```
function Next (Position  : Cursor) return Cursor;
```
55.m/2

In a typical implementation, this will return the next node in a bucket; if Position is the last node in a bucket, this will return the first node in the next nonempty bucket. 55.n/2

A typical implementation will need to a keep a pointer at the map container in the cursor in order to implement this function. 55.o/2

Implementation Note: In: 55.p/2

```
function Find (Container : Map;
               Key       : Key_Type) return Cursor;
```
55.q/2

Find should only compare keys that hash to the same bucket in the hash table. 55.r/2

```
{AI12-0112-1} function Equivalent_Keys (Left, Right : Cursor)
   return Boolean
   with Pre    => (Left /= No_Element and then Right /= No_Element)
                    or else raise Constraint_Error,
        Global => in all;
```
56/5

{*AI95-00302-03*} Equivalent to Equivalent_Keys (Key (Left), Key (Right)). 57/2

```
{AI12-0112-1} function Equivalent_Keys (Left  : Cursor;
                                        Right : Key_Type) return Boolean
   with Pre    => Left /= No_Element or else raise Constraint_Error,
        Global => in all;
```
58/5

{*AI95-00302-03*} Equivalent to Equivalent_Keys (Key (Left), Right). 59/2

```
{AI12-0112-1} function Equivalent_Keys (Left  : Key_Type;
                                        Right : Cursor) return Boolean
   with Pre    => Right /= No_Element or else raise Constraint_Error,
        Global => in all;
```
60/5

{*AI95-00302-03*} Equivalent to Equivalent_Keys (Left, Key (Right)). 61/2

```
{AI12-0112-1} {AI12-0266-1} function Iterate (Container : in Map)
   return Map_Iterator_Interfaces.Parallel_Iterator'Class
   with Post => Tampering_With_Cursors_Prohibited (Container);
```
61.1/5

{*AI05-0212-1*} {*AI05-0265-1*} {*AI05-0269-1*} {*AI12-0266-1*} Iterate returns an iterator object (see 5.5.1) that will generate a value for a loop parameter (see 5.5.2) designating each node in Container, starting with the first node and moving the cursor according to the successor relation when used as a forward iterator, and processing all nodes concurrently when used as a parallel iterator. Tampering with the cursors of Container is prohibited while the iterator object exists (in 61.2/5

particular, in the sequence_of_statements of the loop_statement whose iterator_specification denotes this object). The iterator object needs finalization.

Implementation Advice

62/2 {*AI95-00302-03*} If *N* is the length of a map, the average time complexity of the subprograms Element, Insert, Include, Replace, Delete, Exclude, and Find that take a key parameter should be *O*(log *N*). The average time complexity of the subprograms that take a cursor parameter should be *O*(1). The average time complexity of Reserve_Capacity should be *O*(*N*).

62.a/2 **Implementation Advice:** The average time complexity of Element, Insert, Include, Replace, Delete, Exclude, and Find operations that take a key parameter for Containers.Hashed_Maps should be *O*(log *N*). The average time complexity of the subprograms of Containers.Hashed_Maps that take a cursor parameter should be *O*(1). The average time complexity of Containers.Hashed_Maps.Reserve_Capacity should be *O*(*N*).

62.b/2 **Reason:** We do not mean to overly constrain implementation strategies here. However, it is important for portability that the performance of large containers has roughly the same factors on different implementations. If a program is moved to an implementation for which Find is *O*(*N*), that program could be unusable when the maps are large. We allow *O*(log *N*) access because the proportionality constant and caching effects are likely to be larger than the log factor, and we don't want to discourage innovative implementations.

Extensions to Ada 95

62.c/2 {*AI95-00302-03*} The generic package Containers.Hashed_Maps is new.

Incompatibilities With Ada 2005

62.d/3 {*AI05-0001-1*} Subprograms Assign and Copy are added to Containers.Hashed_Maps. If an instance of Containers.Hashed_Maps is referenced in a use_clause, and an entity *E* with the same defining_identifier as a new entity in Containers.Hashed_Maps is defined in a package that is also referenced in a use_clause, the entity *E* may no longer be use-visible, resulting in errors. This should be rare and is easily fixed if it does occur.

Extensions to Ada 2005

62.e/3 {*AI05-0212-1*} Added iterator and indexing support to make hashed map containers more convenient to use.

Wording Changes from Ada 2005

62.f/3 {*AI05-0084-1*} **Correction:** Added a pragma Remote_Types so that containers can be used in distributed programs.

Incompatibilities With Ada 2012

62.g/5 {*AI12-0111-1*} {*AI12-0112-1*} {*AI12-0339-1*} A number of new subprograms, types, and even a nested package were added to Containers.Hashed_Maps to better support contracts and stable views. Therefore, a use clause conflict is possible; see the introduction of Annex A for more on this topic.

Extensions to Ada 2012

62.h/5 {*AI12-0212-1*} Maps now support named container aggregates, so aggregate syntax can be used to create Maps.

62.i/5 {*AI12-0266-1*} The iterator for the container now can return a parallel iterator which can be used to process the container in parallel.

Wording Changes from Ada 2012

62.j/5 {*AI12-0112-1*} Added contracts to this container. This includes describing some of the semantics with pre- and postconditions, rather than English text. Note that the preconditions can be Suppressed (see 11.5).

A.18.6 The Generic Package Containers.Ordered_Maps

Static Semantics

{*AI95-00302-03*} The generic library package Containers.Ordered_Maps has the following declaration: 1/2

```
{AI05-0084-1} {AI05-0212-1} {AI12-0112-1} with Ada.Iterator_Interfaces;                    2/5
generic
   type Key_Type is private;
   type Element_Type is private;
   with function "<" (Left, Right : Key_Type) return Boolean is <>;
   with function "=" (Left, Right : Element_Type) return Boolean is <>;
package Ada.Containers.Ordered_Maps
   with Preelaborate, Remote_Types,
        Nonblocking, Global => in out synchronized is
```

Discussion: {*AI12-0112-1*} For discussion on the reasons and meaning of the specifications of the Global and Nonblocking aspects in this generic package, see the notes on the equivalent operations in the specification of the Containers.Vectors package (see A.18.2). 2.a/5

```
{AI12-0112-1}    function Equivalent_Keys (Left, Right : Key_Type) return Boolean          3/5
      is (not ((Left < Right) or (Right < Left)));

{AI05-0212-1} {AI12-0111-1} {AI12-0112-1} {AI12-0212-1} {AI12-0339-1} {AI12-0399-1}       4/5
type Map is tagged private
      with Constant_Indexing => Constant_Reference,
           Variable_Indexing => Reference,
           Default_Iterator  => Iterate,
           Iterator_Element  => Element_Type,
           Iterator_View     => Stable.Map,
           Aggregate         => (Empty     => Empty,
                                 Add_Named => Insert),
           Stable_Properties => (Length,
                                 Tampering_With_Cursors_Prohibited,
                                 Tampering_With_Elements_Prohibited),
           Default_Initial_Condition =>
              Length (Map) = 0 and then
              (not Tampering_With_Cursors_Prohibited (Map)) and then
              (not Tampering_With_Elements_Prohibited (Map)),
           Preelaborable_Initialization;

{AI12-0399-1}    type Cursor is private                                                    5/5
      with Preelaborable_Initialization;

   Empty_Map : constant Map;                                                             6/2

   No_Element : constant Cursor;                                                         7/2

{AI05-0212-1} {AI12-0112-1}    function Has_Element (Position : Cursor) return          7.1/5
Boolean
      with Nonblocking, Global => in all, Use_Formal => null;

{AI12-0112-1}    function Has_Element (Container : Map; Position : Cursor)               7.2/5
      return Boolean
      with Nonblocking, Global => null, Use_Formal => null;

{AI05-0212-1}    package Map_Iterator_Interfaces is new                                  7.3/5
       Ada.Iterator_Interfaces (Cursor, Has_Element);

   function "=" (Left, Right : Map) return Boolean;                                      8/2

{AI12-0112-1}    function Tampering_With_Cursors_Prohibited                              8.1/5
      (Container : Map) return Boolean
      with Nonblocking, Global => null, Use_Formal => null;

{AI12-0112-1}    function Tampering_With_Elements_Prohibited                             8.2/5
      (Container : Map) return Boolean
      with Nonblocking, Global => null, Use_Formal => null;
```

8.3/5
```
{AI12-0339-1}    function Empty return Map
      is (Empty_Map)
      with Post =>
             not Tampering_With_Elements_Prohibited (Empty'Result) and then
             not Tampering_With_Cursors_Prohibited (Empty'Result) and then
             Length (Empty'Result) = 0;
```

9/5
```
{AI12-0112-1}    function Length (Container : Map) return Count_Type
      with Nonblocking, Global => null, Use_Formal => null;
```

10/5
```
{AI12-0112-1}    function Is_Empty (Container : Map) return Boolean
      with Nonblocking, Global => null, Use_Formal => null,
           Post => Is_Empty'Result = (Length (Container) = 0);
```

11/5
```
{AI12-0112-1}    procedure Clear (Container : in out Map)
      with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                       or else raise Program_Error,
           Post => Length (Container) = 0;
```

12/5
```
{AI12-0112-1}    function Key (Position : Cursor) return Key_Type
      with Pre  => Position /= No_Element or else raise Constraint_Error,
           Nonblocking, Global => in all, Use_Formal => Key_Type;
```

12.1/5
```
{AI12-0112-1}    function Key (Container : Map;
                               Position : Cursor) return Key_Type
      with Pre  => (Position /= No_Element
                       or else raise Constraint_Error) and then
                     (Has_Element (Container, Position)
                       or else raise Program_Error),
           Nonblocking, Global => null, Use_Formal => Key_Type;
```

13/5
```
{AI12-0112-1}    function Element (Position : Cursor) return Element_Type
      with Pre  => Position /= No_Element or else raise Constraint_Error,
           Nonblocking, Global => null, Use_Formal => Element_Type;
```

13.1/5
```
{AI12-0112-1}    function Element (Container : Map;
                                   Position  : Cursor) return Element_Type
      with Pre  => (Position /= No_Element
                       or else raise Constraint_Error) and then
                     (Has_Element (Container, Position)
                       or else raise Program_Error),
           Nonblocking, Global => null, Use_Formal => Element_Type;
```

14/5
```
{AI12-0112-1}    procedure Replace_Element (Container : in out Map;
                                            Position  : in     Cursor;
                                            New_item  : in     Element_Type)
      with Pre  => (not Tampering_With_Elements_Prohibited (Container)
                       or else raise Program_Error) and then
                     (Position /= No_Element
                       or else raise Constraint_Error) and then
                     (Has_Element (Container, Position)
                       or else raise Program_Error);
```

15/5
```
{AI12-0112-1}    procedure Query_Element
     (Position : in Cursor;
      Process  : not null access procedure (Key     : in Key_Type;
                                            Element : in Element_Type))
      with Pre  => Position /= No_Element
                       or else raise Constraint_Error,
           Global => in all;
```

15.1/5
```
{AI12-0112-1}    procedure Query_Element
     (Container : in Map;
      Position  : in Cursor;
      Process   : not null access procedure (Key     : in Key_Type;
                                             Element : in Element_Type))
      with Pre  => (Position /= No_Element
                       or else raise Constraint_Error) and then
                    (Has_Element (Container, Position)
                       or else raise Program_Error);
```

```
{AI12-0112-1}    procedure Update_Element                                                16/5
     (Container : in out Map;
      Position  : in     Cursor;
      Process   : not null access procedure
                       (Key     : in     Key_Type;
                        Element : in out Element_Type))
      with Pre  => (Position /= No_Element
                       or else raise Constraint_Error) and then
                    (Has_Element (Container, Position)
                       or else raise Program_Error);

{AI05-0212-1} {AI12-0112-1}    type Constant_Reference_Type                             16.1/5
         (Element : not null access constant Element_Type) is private
      with Implicit_Dereference => Element,
           Nonblocking, Global => in out synchronized,
           Default_Initial_Condition => (raise Program_Error);

{AI05-0212-1} {AI12-0112-1}    type Reference_Type (Element : not null access          16.2/5
Element_Type) is private
      with Implicit_Dereference => Element,
           Nonblocking, Global => in out synchronized,
           Default_Initial_Condition => (raise Program_Error);

{AI05-0212-1} {AI12-0112-1}    function Constant_Reference (Container : aliased in     16.3/5
Map;
                                Position  : in Cursor)
      return Constant_Reference_Type
      with Pre  => (Position /= No_Element
                       or else raise Constraint_Error) and then
                    (Has_Element (Container, Position)
                       or else raise Program_Error),
           Post => Tampering_With_Cursors_Prohibited (Container),
           Nonblocking, Global => null, Use_Formal => null;

{AI05-0212-1} {AI12-0112-1}    function Reference (Container : aliased in out Map;     16.4/5
                        Position  : in Cursor)
      return Reference_Type
      with Pre  => (Position /= No_Element
                       or else raise Constraint_Error) and then
                    (Has_Element (Container, Position)
                       or else raise Program_Error),
           Post => Tampering_With_Cursors_Prohibited (Container),
           Nonblocking, Global => null, Use_Formal => null;

{AI05-0212-1} {AI12-0112-1}    function Constant_Reference (Container : aliased in     16.5/5
Map;
                                Key       : in Key_Type)
      return Constant_Reference_Type
      with Pre  => Find (Container, Key) /= No_Element
                       or else raise Constraint_Error,
           Post => Tampering_With_Cursors_Prohibited (Container),
           Nonblocking, Global => null, Use_Formal => null;

{AI05-0212-1} {AI12-0112-1}    function Reference (Container : aliased in out Map;     16.6/5
                        Key       : in Key_Type)
      return Reference_Type
      with Pre  => Find (Container, Key) /= No_Element
                       or else raise Constraint_Error,
           Post => Tampering_With_Cursors_Prohibited (Container),
           Nonblocking, Global => null, Use_Formal => null;

{AI05-0001-1} {AI12-0112-1}    procedure Assign (Target : in out Map; Source : in      16.7/5
Map)
      with Pre  => not Tampering_With_Cursors_Prohibited (Target)
                       or else raise Program_Error,
           Post => Length (Source) = Length (Target);
```

16.8/5
```
{AI05-0001-1} {AI12-0112-1}    function Copy (Source : Map)
      return Map
      with Post =>
         Length (Copy'Result) = Length (Source) and then
         not Tampering_With_Elements_Prohibited (Copy'Result) and then
         not Tampering_With_Cursors_Prohibited (Copy'Result);
```

17/5
```
{AI12-0112-1}    procedure Move (Target : in out Map;
                                Source : in out Map)
      with Pre  => (not Tampering_With_Cursors_Prohibited (Target)
                       or else raise Program_Error) and then
                   (not Tampering_With_Cursors_Prohibited (Source)
                       or else raise Program_Error),
           Post => (if not Target'Has_Same_Storage (Source) then
                       Length (Target) = Length (Source'Old) and then
                       Length (Source) = 0);
```

18/5
```
{AI12-0112-1}    procedure Insert (Container : in out Map;
                                   Key       : in     Key_Type;
                                   New_Item  : in     Element_Type;
                                   Position  :    out Cursor;
                                   Inserted  :    out Boolean)
      with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                       or else raise Program_Error) and then
                   (Length (Container) <= Count_Type'Last - 1
                       or else raise Constraint_Error),
           Post => (declare
                      Original_Length : constant Count_Type :=
                         Length (Container)'Old;
                    begin
                      Has_Element (Container, Position) and then
                      (if Inserted then
                         Length (Container) = Original_Length + 1
                       else
                         Length (Container) = Original_Length));
```

19/5
```
{AI12-0112-1}    procedure Insert (Container : in out Map;
                                   Key       : in     Key_Type;
                                   Position  :    out Cursor;
                                   Inserted  :    out Boolean)
      with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                       or else raise Program_Error) and then
                   (Length (Container) <= Count_Type'Last - 1
                       or else raise Constraint_Error),
           Post => (declare
                      Original_Length : constant Count_Type :=
                         Length (Container)'Old;
                    begin
                      Has_Element (Container, Position) and then
                      (if Inserted then
                         Length (Container) = Original_Length + 1
                       else
                         Length (Container) = Original_Length));
```

20/5
```
{AI12-0112-1}    procedure Insert (Container : in out Map;
                                   Key       : in     Key_Type;
                                   New_Item  : in     Element_Type)
      with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                       or else raise Program_Error) and then
                   (Length (Container) <= Count_Type'Last - 1
                       or else raise Constraint_Error),
           Post => Length (Container) = Length (Container)'Old + 1;
```

```
{AI12-0112-1}    procedure Include (Container : in out Map;                                    21/5
                                   Key       : in     Key_Type;
                                   New_Item  : in     Element_Type)
                 with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                                   or else raise Program_Error) and then
                              (Length (Container) <= Count_Type'Last - 1
                                   or else raise Constraint_Error),
                      Post => (declare
                                  Original_Length : constant Count_Type :=
                                     Length (Container)'Old;
                               begin
                                  Length (Container)
                                     in Original_Length | Original_Length + 1);

{AI12-0112-1}    procedure Replace (Container : in out Map;                                   22/5
                                   Key       : in     Key_Type;
                                   New_Item  : in     Element_Type)
                 with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                                   or else raise Program_Error,
                      Post => Length (Container) = Length (Container)'Old;

{AI12-0112-1}    procedure Exclude (Container : in out Map;                                   23/5
                                   Key       : in     Key_Type)
                 with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                                   or else raise Program_Error,
                      Post => (declare
                                  Original_Length : constant Count_Type :=
                                     Length (Container)'Old;
                               begin
                                  Length (Container)
                                     in Original_Length - 1 | Original_Length);

{AI12-0112-1}    procedure Delete (Container : in out Map;                                    24/5
                                  Key       : in     Key_Type)
                 with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                                   or else raise Program_Error,
                      Post => Length (Container) = Length (Container)'Old - 1;

{AI12-0112-1}    procedure Delete (Container : in out Map;                                    25/5
                                   Position  : in out Cursor)
                 with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                                   or else raise Program_Error) and then
                              (Position /= No_Element
                                   or else raise Constraint_Error) and then
                              (Has_Element (Container, Position)
                                 or else raise Program_Error),
                      Post => Length (Container) = Length (Container)'Old - 1 and then
                              Position = No_Element;

{AI12-0112-1}    procedure Delete_First (Container : in out Map)                               26/5
                 with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                                   or else raise Program_Error,
                      Post => (declare
                                  Original_Length : constant Count_Type :=
                                     Length (Container)'Old;
                               begin
                                 (if Original_Length = 0 then Length (Container) = 0
                                  else Length (Container) = Original_Length - 1));

{AI12-0112-1}    procedure Delete_Last (Container : in out Map)                               27/5
                 with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                                   or else raise Program_Error,
                      Post => (declare
                                  Original_Length : constant Count_Type :=
                                     Length (Container)'Old;
                               begin
                                 (if Original_Length = 0 then Length (Container) = 0
                                  else Length (Container) = Original_Length - 1));
```

28/5 {*AI12-0112-1*} **function** First (Container : Map) **return** Cursor
with Nonblocking, Global => **null**, Use_Formal => **null**,
Post => (**if not** Is_Empty (Container)
then Has_Element (Container, First'Result)
else First'Result = No_Element);

29/5 {*AI12-0112-1*} **function** First_Element (Container : Map) **return** Element_Type
with Pre => (**not** Is_Empty (Container)
or else raise Constraint_Error);

30/5 {*AI12-0112-1*} **function** First_Key (Container : Map) **return** Key_Type
with Pre => (**not** Is_Empty (Container)
or else raise Constraint_Error);

31/5 {*AI12-0112-1*} **function** Last (Container : Map) **return** Cursor
with Nonblocking, Global => **null**, Use_Formal => **null**,
Post => (**if** not Is_Empty (Container)
then Has_Element (Container, Last'Result)
else Last'Result = No_Element);

32/5 {*AI12-0112-1*} **function** Last_Element (Container : Map) **return** Element_Type
with Pre => (**not** Is_Empty (Container)
or else raise Constraint_Error);

33/5 {*AI12-0112-1*} **function** Last_Key (Container : Map) **return** Key_Type
with Pre => (**not** Is_Empty (Container)
or else raise Constraint_Error);

34/5 {*AI12-0112-1*} **function** Next (Position : Cursor) **return** Cursor
with Nonblocking, Global => **in all**, Use_Formal => **null**,
Post => (**if** Position = No_Element **then** Next'Result = No_Element);

34.1/5 {*AI12-0112-1*} **function** Next (Container : Map;
Position : Cursor) **return** Cursor
with Nonblocking, Global => **null**, Use_Formal => **null**,
Pre => Position = No_Element **or else**
Has_Element (Container, Position)
or else raise Program_Error,
Post => (**if** Position = No_Element **then** Next'Result = No_Element
elsif Next'Result = No_Element **then**
Position = Last (Container)
else Has_Element (Container, Next'Result));

35/5 {*AI12-0112-1*} **procedure** Next (Position : **in out** Cursor)
with Nonblocking, Global => **in all**, Use_Formal => **null**;

35.1/5 {*AI12-0112-1*} **procedure** Next (Container : **in** Map;
Position : **in out** Cursor)
with Nonblocking, Global => **null**, Use_Formal => **null**,
Pre => Position = No_Element **or else**
Has_Element (Container, Position)
or else raise Program_Error,
Post => (**if** Position /= No_Element
then Has_Element (Container, Position));

36/5 {*AI12-0112-1*} **function** Previous (Position : Cursor) **return** Cursor
with Nonblocking, Global => **in all**, Use_Formal => **null**,
Post => (**if** Position = No_Element **then**
Previous'Result = No_Element);

36.1/5 {*AI12-0112-1*} **function** Previous (Container : Map;
Position : Cursor) **return** Cursor
with Nonblocking, Global => **null**, Use_Formal => **null**,
Pre => Position = No_Element **or else**
Has_Element (Container, Position)
or else raise Program_Error,
Post => (**if** Position = No_Element **then**
Previous'Result = No_Element
elsif Previous'Result = No_Element **then**
Position = First (Container)
else Has_Element (Container, Previous'Result));

37/5 {*AI12-0112-1*} **procedure** Previous (Position : **in out** Cursor)
with Nonblocking, Global => **in all**, Use_Formal => **null**;

```
{AI12-0112-1}    procedure Previous (Container : in     Map;                    37.1/5
                                    Position  : in out Cursor)
      with Nonblocking, Global => null, Use_Formal => null,
           Pre  => Position = No_Element or else
                   Has_Element (Container, Position)
                      or else raise Program_Error,
           Post => (if Position /= No_Element
                    then Has_Element (Container, Position));

{AI12-0112-1}    function Find (Container : Map;                                  38/5
                                Key       : Key_Type) return Cursor
      with Post => (if Find'Result /= No_Element
                    then Has_Element (Container, Find'Result));

   function Element (Container : Map;                                            39/2
                     Key       : Key_Type) return Element_Type;

{AI12-0112-1}    function Floor (Container : Map;                                 40/5
                                 Key       : Key_Type) return Cursor
      with Post => (if Floor'Result /= No_Element
                    then Has_Element (Container, Floor'Result));

{AI12-0112-1}    function Ceiling (Container : Map;                               41/5
                                   Key       : Key_Type) return Cursor
      with Post => (if Ceiling'Result /= No_Element
                    then Has_Element (Container, Ceiling'Result));

   function Contains (Container : Map;                                           42/2
                      Key       : Key_Type) return Boolean;

This paragraph was deleted.{AI05-0212-1}                                         43/3

{AI12-0112-1}    function "<" (Left, Right : Cursor) return Boolean              44/5
      with Pre    => (Left /= No_Element and then Right /= No_Element)
                        or else raise Constraint_Error,
           Global => in all;

{AI12-0112-1}    function ">" (Left, Right : Cursor) return Boolean              45/5
      with Pre    => (Left /= No_Element and then Right /= No_Element)
                        or else raise Constraint_Error,
           Global => in all;

{AI12-0112-1}    function "<" (Left : Cursor; Right : Key_Type) return Boolean   46/5
      with Pre    => Left /= No_Element or else raise Constraint_Error,
           Global => in all;

{AI12-0112-1}    function ">" (Left : Cursor; Right : Key_Type) return Boolean   47/5
      with Pre    => Left /= No_Element or else raise Constraint_Error,
           Global => in all;

{AI12-0112-1}    function "<" (Left : Key_Type; Right : Cursor) return Boolean   48/5
      with Pre    => Right /= No_Element or else raise Constraint_Error,
           Global => in all;

{AI12-0112-1}    function ">" (Left : Key_Type; Right : Cursor) return Boolean   49/5
      with Pre    => Right /= No_Element or else raise Constraint_Error,
           Global => in all;

{AI12-0112-1}    procedure  Iterate                                              50/5
     (Container : in Map;
      Process   : not null access procedure (Position : in Cursor))
      with Allows_Exit;

{AI12-0112-1}    procedure Reverse_Iterate                                       51/5
     (Container : in Map;
      Process   : not null access procedure (Position : in Cursor))
      with Allows_Exit;

{AI05-0212-1} {AI12-0112-1} {AI12-0266-1}    function Iterate (Container : in Map)   51.1/5
      return Map_Iterator_Interfaces.Parallel_Reversible_Iterator'Class
      with Post => Tampering_With_Cursors_Prohibited (Container);
```

51.2/5
```
{AI05-0262-1} {AI12-0112-1}    function Iterate (Container : in Map; Start : in
Cursor)
      return Map_Iterator_Interfaces.Reversible_Iterator'Class
      with Pre  => (Start /= No_Element
                      or else raise Constraint_Error) and then
                   (Has_Element (Container, Start)
                      or else raise Program_Error),
           Post => Tampering_With_Cursors_Prohibited (Container);
```

51.3/5
```
{AI12-0111-1}    package Stable is
```

51.4/5
```
{AI12-0111-1} {AI12-0339-1} {AI12-0399-1} {AI12-0407-1}       type Map (Base : not
null access Ordered_Maps.Map) is
         tagged limited private
         with Constant_Indexing => Constant_Reference,
              Variable_Indexing => Reference,
              Default_Iterator  => Iterate,
              Iterator_Element  => Element_Type,
              Stable_Properties => (Length),
              Global            => null,
              Default_Initial_Condition => Length (Map) = 0,
              Preelaborable_Initialization;
```

51.5/5
```
{AI12-0111-1} {AI12-0399-1}       type Cursor is private
         with Preelaborable_Initialization;
```

51.6/5
```
{AI12-0111-1}       Empty_Map : constant Map;
```

51.7/5
```
{AI12-0111-1}       No_Element : constant Cursor;
```

51.8/5
```
{AI12-0111-1}       function Has_Element (Position : Cursor) return Boolean
         with Nonblocking, Global => in all, Use_Formal => null;
```

51.9/5
```
{AI12-0111-1}       package Map_Iterator_Interfaces is new
         Ada.Iterator_Interfaces (Cursor, Has_Element);
```

51.10/5
```
{AI12-0111-1}       procedure Assign (Target : in out Ordered_Maps.Map;
                                      Source : in Map)
         with Post => Length (Source) = Length (Target);
```

51.11/5
```
{AI12-0111-1}       function Copy (Source : Ordered_Maps.Map) return Map
         with Post => Length (Copy'Result) = Length (Source);
```

51.12/5
```
{AI12-0111-1}       type Constant_Reference_Type
            (Element : not null access constant Element_Type) is private
         with Implicit_Dereference => Element,
              Nonblocking, Global => null, Use_Formal => null,
              Default_Initial_Condition => (raise Program_Error);
```

51.13/5
```
{AI12-0111-1}       type Reference_Type
            (Element : not null access Element_Type) is private
         with Implicit_Dereference => Element,
              Nonblocking, Global => null, Use_Formal => null,
              Default_Initial_Condition => (raise Program_Error);
```

51.14/5
```
{AI12-0111-1}       -- Additional subprograms as described in the text
      -- are declared here.
```

51.15/5
```
{AI12-0111-1}    private
```

51.16/5
```
{AI12-0111-1}       ... -- not specified by the language
```

51.17/5
```
{AI12-0111-1}    end Stable;
```

52/2
```
private
```

53/2
```
   ... -- not specified by the language
```

54/2
```
end Ada.Containers.Ordered_Maps;
```

55/2 {*AI95-00302-03*} Two keys *K1* and *K2* are *equivalent* if both *K1 < K2* and *K2 < K1* return False, using the generic formal "<" operator for keys. Function Equivalent_Keys returns True if Left and Right are equivalent, and False otherwise.

{*AI95-00302-03*} {*AI05-0044-1*} The actual function for the generic formal function "<" on Key_Type values is expected to return the same value each time it is called with a particular pair of key values. It should define a strict weak ordering relationship (see A.18). If the actual for "<" behaves in some other manner, the behavior of this package is unspecified. Which subprograms of this package call "<" and how many times they call it, is unspecified. 56/3

Implementation Note: The implementation is not required to protect against "<" raising an exception, or returning random results, or any other “bad” behavior. It's not practical to do so, and a broken "<" function makes the container unusable. 56.a/2

The implementation can call "<" whenever it is needed; we don't want to specify how often that happens. The result must remain the same (this is a logically pure function), or the behavior is unspecified. 56.b/2

{*AI95-00302-03*} If the value of a key stored in a map is changed other than by an operation in this package such that at least one of "<" or "=" give different results, the behavior of this package is unspecified. 57/2

Implementation Note: The implementation is not required to protect against changes to key values other than via the operations declared in the Ordered_Maps package. 57.a/2

To see how this could happen, imagine an instance of Ordered_Maps package where the key type is an access-to-variable type and "<" returns a value derived from comparing the components of the designated objects. Then, any operation that has a key value (even if the key value is constant) could modify those components and change the result of "<": 57.b/2

```
Key (Map).Some_Component := New_Value;
```
57.c/2

This is really a design error on the part of the user of the map; it shouldn't be possible to modify keys stored in a map such that "<" changes. But we can't prevent this error anymore than we can prevent someone passing as "<" a routine that produces random answers. 57.d/2

{*AI95-00302-03*} {*AI05-0262-1*} The *first node* of a nonempty map is the one whose key is less than the key of all the other nodes in the map. The *last node* of a nonempty map is the one whose key is greater than the key of all the other elements in the map. The *successor* of a node is the node with the smallest key that is larger than the key of the given node. The *predecessor* of a node is the node with the largest key that is smaller than the key of the given node. All comparisons are done using the generic formal "<" operator for keys. 58/3

```
{AI12-0112-1} function Copy (Source : Map)
   return Map
   with Post =>
      Length (Copy'Result) = Length (Source) and then
      not Tampering_With_Elements_Prohibited (Copy'Result) and then
      not Tampering_With_Cursors_Prohibited (Copy'Result);
```
58.1/5

{*AI05-0001-1*} Returns a map whose keys and elements are initialized from the corresponding keys and elements of Source. 58.2/3

```
{AI12-0112-1} procedure Delete_First (Container : in out Map)
   with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error,
        Post => (declare
                    Original_Length : constant Count_Type :=
                       Length (Container)'Old;
                 begin
                    (if Original_Length = 0 then Length (Container) = 0
                     else Length (Container) = Original_Length - 1));
```
59/5

{*AI95-00302-03*} {*AI05-0264-1*} If Container is empty, Delete_First has no effect. Otherwise, the node designated by First (Container) is removed from Container. Delete_First tampers with the cursors of Container. 60/3

61/5
```
{AI12-0112-1} procedure Delete_Last (Container : in out Map)
   with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error,
        Post => (declare
                   Original_Length : constant Count_Type :=
                      Length (Container)'Old;
                 begin
                   (if Original_Length = 0 then Length (Container) = 0
                    else Length (Container) = Original_Length - 1));
```

62/3 {*AI95-00302-03*} {*AI05-0264-1*} If Container is empty, Delete_Last has no effect. Otherwise, the node designated by Last (Container) is removed from Container. Delete_Last tampers with the cursors of Container.

63/5
```
{AI12-0112-1} function First_Element (Container : Map) return Element_Type
   with Pre => (not Is_Empty (Container)
                    or else raise Constraint_Error);
```

64/2 {*AI95-00302-03*} Equivalent to Element (First (Container)).

65/5
```
{AI12-0112-1} function First_Key (Container : Map) return Key_Type
   with Pre => (not Is_Empty (Container)
                    or else raise Constraint_Error);
```

66/2 {*AI95-00302-03*} Equivalent to Key (First (Container)).

67/5
```
{AI12-0112-1} function Last (Container : Map) return Cursor
   with Nonblocking, Global => null, Use_Formal => null,
        Post => (if not Is_Empty (Container)
                 then Has_Element (Container, Last'Result)
                 else Last'Result = No_Element);
```

68/2 {*AI95-00302-03*} Returns a cursor that designates the last node in Container. If Container is empty, returns No_Element.

69/5
```
{AI12-0112-1} function Last_Element (Container : Map) return Element_Type
   with Pre => (not Is_Empty (Container)
                    or else raise Constraint_Error);
```

70/2 {*AI95-00302-03*} Equivalent to Element (Last (Container)).

71/5
```
{AI12-0112-1} function Last_Key (Container : Map) return Key_Type
   with Pre => (not Is_Empty (Container)
                    or else raise Constraint_Error);
```

72/2 {*AI95-00302-03*} Equivalent to Key (Last (Container)).

73/5
```
{AI12-0112-1} function Previous (Position : Cursor) return Cursor
   with Nonblocking, Global => in all, Use_Formal => null,
        Post => (if Position = No_Element then
                    Previous'Result = No_Element);
```

74/3 {*AI95-00302-03*} {*AI05-0262-1*} If Position equals No_Element, then Previous returns No_Element. Otherwise, Previous returns a cursor designating the predecessor node of the one designated by Position. If Position designates the first element, then Previous returns No_Element.

```
function Previous (Container : Map;                                                    74.1/5
                   Position : Cursor) return Cursor
   with Nonblocking, Global => null, Use_Formal => null,
        Pre  => Position = No_Element or else
                Has_Element (Container, Position)
                   or else raise Program_Error,
        Post => (if Position = No_Element then
                   Previous'Result = No_Element
                 elsif Previous'Result = No_Element then
                   Position = First (Container)
                 else Has_Element (Container, Previous'Result));
```

{*AI12-0112-1*} Returns a cursor designating the predecessor of the node designated by Position in Container, if any. 74.2/5

```
{AI12-0112-1} procedure Previous (Position : in out Cursor)                            75/5
   with Nonblocking, Global => in all, Use_Formal => null;
```

{*AI95-00302-03*} Equivalent to Position := Previous (Position). 76/2

```
procedure Previous (Container : in     Map;                                            76.1/5
                    Position  : in out Cursor)
   with Nonblocking, Global => null, Use_Formal => null,
        Pre  => Position = No_Element or else
                Has_Element (Container, Position)
                   or else raise Program_Error,
        Post => (if Position /= No_Element
                 then Has_Element (Container, Position));
```

{*AI12-0112-1*} Equivalent to Position := Previous (Container, Position). 76.2/5

```
{AI12-0112-1} function Floor (Container : Map;                                         77/5
                  Key       : Key_Type) return Cursor
   with Post => (if Floor'Result /= No_Element
                 then Has_Element (Container, Floor'Result));
```

{*AI95-00302-03*} {*AI05-0264-1*} Floor searches for the last node whose key is not greater than Key, using the generic formal "<" operator for keys. If such a node is found, a cursor that designates it is returned. Otherwise, No_Element is returned. 78/3

```
{AI12-0112-1} function Ceiling (Container : Map;                                       79/5
                    Key       : Key_Type) return Cursor
   with Post => (if Ceiling'Result /= No_Element
                 then Has_Element (Container, Ceiling'Result));
```

{*AI95-00302-03*} {*AI05-0264-1*} Ceiling searches for the first node whose key is not less than Key, using the generic formal "<" operator for keys. If such a node is found, a cursor that designates it is returned. Otherwise, No_Element is returned. 80/3

```
{AI12-0112-1} function "<" (Left, Right : Cursor) return Boolean                       81/5
   with Pre    => (Left /= No_Element and then Right /= No_Element)
                     or else raise Constraint_Error,
        Global => in all;
```

{*AI95-00302-03*} Equivalent to Key (Left) < Key (Right). 82/2

```
{AI12-0112-1} function ">" (Left, Right : Cursor) return Boolean                       83/5
   with Pre    => (Left /= No_Element and then Right /= No_Element)
                     or else raise Constraint_Error,
        Global => in all;
```

{*AI95-00302-03*} Equivalent to Key (Right) < Key (Left). 84/2

85/5
```
{AI12-0112-1} function "<" (Left : Cursor; Right : Key_Type) return Boolean
   with Pre    => Left /= No_Element or else raise Constraint_Error,
        Global => in all;
```

86/2 {*AI95-00302-03*} Equivalent to Key (Left) < Right.

87/5
```
{AI12-0112-1} function ">" (Left : Cursor; Right : Key_Type) return Boolean
   with Pre    => Left /= No_Element or else raise Constraint_Error,
           Global => in all;
```

88/2 {*AI95-00302-03*} Equivalent to Right < Key (Left).

89/5
```
{AI12-0112-1} function "<" (Left : Key_Type; Right : Cursor) return Boolean
   with Pre    => Right /= No_Element or else raise Constraint_Error,
           Global => in all;
```

90/2 {*AI95-00302-03*} Equivalent to Left < Key (Right).

91/5
```
{AI12-0112-1} function ">" (Left : Key_Type; Right : Cursor) return Boolean
   with Pre    => Right /= No_Element or else raise Constraint_Error,
           Global => in all;
```

92/2 {*AI95-00302-03*} Equivalent to Key (Right) < Left.

93/5
```
{AI12-0112-1} procedure Reverse_Iterate
  (Container : in Map;
   Process   : not null access procedure (Position : in Cursor))
   with Allows_Exit;
```

94/3 {*AI95-00302-03*} {*AI05-0212-1*} Iterates over the nodes in Container as per procedure Iterate, with the difference that the nodes are traversed in predecessor order, starting with the last node.

94.1/5
```
{AI12-0112-1} {AI12-0266-1} function Iterate (Container : in Map)
   return Map_Iterator_Interfaces.Parallel_Reversible_Iterator'Class
   with Post => Tampering_With_Cursors_Prohibited (Container);
```

94.2/5 {*AI05-0212-1*} {*AI05-0265-1*} {*AI05-0269-1*} {*AI12-0266-1*} Iterate returns an iterator object (see 5.5.1) that will generate a value for a loop parameter (see 5.5.2) designating each node in Container, starting with the first node and moving the cursor according to the successor relation when used as a forward iterator, and starting with the last node and moving the cursor according to the predecessor relation when used as a reverse iterator, and processing all nodes concurrently when used as a parallel iterator. Tampering with the cursors of Container is prohibited while the iterator object exists (in particular, in the sequence_of_statements of the loop_statement whose iterator_specification denotes this object). The iterator object needs finalization.

94.3/5
```
function Iterate (Container : in Map; Start : in Cursor)
   return Map_Iterator_Interfaces.Reversible_Iterator'Class
   with Pre  => (Start /= No_Element
                    or else raise Constraint_Error) and then
                (Has_Element (Container, Start)
                    or else raise Program_Error),
        Post => Tampering_With_Cursors_Prohibited (Container);
```

94.4/5 {*AI05-0262-1*} {*AI05-0265-1*} {*AI05-0269-1*} {*AI12-0112-1*} Iterate returns a reversible iterator object (see 5.5.1) that will generate a value for a loop parameter (see 5.5.2) designating each node in Container, starting with the node designated by Start and moving the cursor according to the successor relation when used as a forward iterator, or moving the cursor according to the predecessor relation when used as a reverse iterator. Tampering with the cursors of Container is prohibited while the iterator object exists (in particular, in the sequence_of_statements of the loop_statement whose iterator_specification denotes this object). The iterator object needs finalization.

Discussion: Exits are allowed from the loops created using the iterator objects. In particular, to stop the iteration at a particular cursor, just add 94.a/3

```
exit when Cur = Stop;
```
94.b/3

in the body of the loop (assuming that `Cur` is the loop parameter and `Stop` is the cursor that you want to stop at). 94.c/3

Implementation Advice

{*AI95-00302-03*} If *N* is the length of a map, then the worst-case time complexity of the Element, Insert, Include, Replace, Delete, Exclude, and Find operations that take a key parameter should be *O*((log *N*)**2) or better. The worst-case time complexity of the subprograms that take a cursor parameter should be *O*(1). 95/2

Implementation Advice: The worst-case time complexity of Element, Insert, Include, Replace, Delete, Exclude, and Find operations that take a key parameter for Containers.Ordered_Maps should be *O*((log *N*)**2) or better. The worst-case time complexity of the subprograms of Containers.Ordered_Maps that take a cursor parameter should be *O*(1). 95.a/2

Implementation Note: A balanced (red-black) tree for keys has *O*(log *N*) worst-case performance. Note that a *O*(*N*) worst-case implementation (like a list) would be wrong. 95.b/2

Reason: We do not mean to overly constrain implementation strategies here. However, it is important for portability that the performance of large containers has roughly the same factors on different implementations. If a program is moved to an implementation that takes *O*(*N*) to find elements, that program could be unusable when the maps are large. We allow the extra log *N* factors because the proportionality constant and caching effects are likely to be larger than the log factor, and we don't want to discourage innovative implementations. 95.c/2

Extensions to Ada 95

{*AI95-00302-03*} The generic package Containers.Ordered_Maps is new. 95.d/2

Incompatibilities With Ada 2005

{*AI05-0001-1*} Subprograms Assign and Copy are added to Containers.Ordered_Maps. If an instance of Containers.Ordered_Maps is referenced in a use_clause, and an entity *E* with the same defining_identifier as a new entity in Containers.Ordered_Maps is defined in a package that is also referenced in a use_clause, the entity *E* may no longer be use-visible, resulting in errors. This should be rare and is easily fixed if it does occur. 95.e/3

Extensions to Ada 2005

{*AI05-0212-1*} Added iterator and indexing support to make ordered map containers more convenient to use. 95.f/3

Wording Changes from Ada 2005

{*AI05-0044-1*} **Correction:** Redefined "<" actuals to require a strict weak ordering; the old definition allowed indeterminant comparisons that would not have worked in a container. 95.g/3

{*AI05-0084-1*} **Correction:** Added a pragma Remote_Types so that containers can be used in distributed programs. 95.h/3

Incompatibilities With Ada 2012

{*AI12-0111-1*} {*AI12-0112-1*} {*AI12-0339-1*} A number of new subprograms, types, and even a nested package were added to Containers.Ordered_Maps to better support contracts and stable views. Therefore, a use clause conflict is possible; see the introduction of Annex A for more on this topic. 95.i/5

Extensions to Ada 2012

{*AI12-0212-1*} Maps now support named container aggregates, so aggregate syntax can be used to create Maps. 95.j/5

{*AI12-0266-1*} The iterator for the entire container now can return a parallel iterator which can be used to process the container in parallel. 95.k/5

Wording Changes from Ada 2012

{*AI12-0112-1*} Added contracts to this container. This includes describing some of the semantics with pre- and postconditions, rather than English text. Note that the preconditions can be Suppressed (see 11.5). 95.l/5

A.18.7 Sets

1/2 {*AI95-00302-03*} The language-defined generic packages Containers.Hashed_Sets and Containers.Ordered_Sets provide private types Set and Cursor, and a set of operations for each type. A set container allows elements of an arbitrary type to be stored without duplication. A hashed set uses a hash function to organize elements, while an ordered set orders its element per a specified relation.

2/3 {*AI95-00302-03*} {*AI05-0299-1*} This subclause describes the declarations that are common to both kinds of sets. See A.18.8 for a description of the semantics specific to Containers.Hashed_Sets and A.18.9 for a description of the semantics specific to Containers.Ordered_Sets.

Static Semantics

3/2 {*AI95-00302-03*} The actual function for the generic formal function "=" on Element_Type values is expected to define a reflexive and symmetric relationship and return the same result value each time it is called with a particular pair of values. If it behaves in some other manner, the function "=" on set values returns an unspecified value. The exact arguments and number of calls of this generic formal function by the function "=" on set values are unspecified.

3.a/2 **Ramification:** If the actual function for "=" is not symmetric and consistent, the result returned by the "=" for Set objects cannot be predicted. The implementation is not required to protect against "=" raising an exception, or returning random results, or any other “bad” behavior. And it can call "=" in whatever manner makes sense. But note that only the result of "=" for Set objects is unspecified; other subprograms are not allowed to break if "=" is bad (they aren't expected to use "=").

4/2 {*AI95-00302-03*} The type Set is used to represent sets. The type Set needs finalization (see 7.6).

5/2 {*AI95-00302-03*} A set contains elements. Set cursors designate elements. There exists an equivalence relation on elements, whose definition is different for hashed sets and ordered sets. A set never contains two or more equivalent elements. The *length* of a set is the number of elements it contains.

6/2 {*AI95-00302-03*} Each nonempty set has two particular elements called the *first element* and the *last element* (which may be the same). Each element except for the last element has a *successor element*. If there are no other intervening operations, starting with the first element and repeatedly going to the successor element will visit each element in the set exactly once until the last element is reached. The exact definition of these terms is different for hashed sets and ordered sets.

7/5 {*AI95-00302-03*} {*AI12-0111-1*} {*AI12-0112-1*} [Some operations check for “tampering with cursors” of a container because they depend on the set of elements of the container remaining constant and on elements of the container not being replaced.] When tampering with cursors is *prohibited* for a particular set object *S*, Program_Error is propagated by the finalization of *S*[, as well as by a call that passes *S* to certain of the operations of this package, as indicated by the precondition of such an operation].

7.a/5 **Discussion:** {*AI12-0112-1*} Note that Replace_Element tampers with cursors because it might delete and reinsert the element if it moves in the set. That could change the order of iteration, which is what this check is designed to prevent. Replace also tampers with cursors, as it is defined in terms of Replace_Element.

7.b/5 {*AI12-0112-1*} These inclusions mean that there are no operations that would tamper with elements that do not tamper with cursors. As such, we do not define tampering with elements at all for set containers. Earlier versions of Ada did so just so the description of subprograms are the same between containers, but since we've changed those to pre- and postconditions which are necessarily specific to each container, there no longer seems to be any reason to define tampering with elements for sets.

Paragraphs 8 through 14 are removed as preconditions now describe these rules.

10.a/3 **Ramification:** We don't need to explicitly mention assignment_statement, because that finalizes the target object as part of the operation, and finalization of an object is already defined as tampering with cursors.

15/2 {*AI95-00302-03*} Empty_Set represents the empty Set object. It has a length of 0. If an object of type Set is not otherwise initialized, it is initialized to the same value as Empty_Set.

{*AI95-00302-03*} No_Element represents a cursor that designates no element. If an object of type Cursor is not otherwise initialized, it is initialized to the same value as No_Element. 16/2

{*AI95-00302-03*} {*AI12-0434-1*} The primitive "=" operator for type Cursor returns True if both cursors are No_Element, or designate the same element in the same container. 17/5

To be honest: {*AI12-0434-1*} "The primitive "=" operator" is the one with two parameters of type Cursor which returns Boolean. We're not talking about some other (hidden) primitive function named "=". 17.a/5

{*AI95-00302-03*} Execution of the default implementation of the Input, Output, Read, or Write attribute of type Cursor raises Program_Error. 18/2

Reason: A cursor will probably be implemented in terms of one or more access values, and the effects of streaming access values is unspecified. Rather than letting the user stream junk by accident, we mandate that streaming of cursors raise Program_Error by default. The attributes can always be specified if there is a need to support streaming. 18.a/2

{*AI05-0001-1*} {*AI05-0262-1*} {*AI12-0437-1*} Set'Write for a Set object *S* writes Length(*S*) elements of the set to the stream. It may also write additional information about the set. 18.1/5

{*AI05-0001-1*} {*AI05-0262-1*} Set'Read reads the representation of a set from the stream, and assigns to *Item* a set with the same length and elements as was written by Set'Write. 18.2/3

Ramification: Streaming more elements than the container length is wrong. For implementation implications of this rule, see the Implementation Note in A.18.2. 18.b/3

```
{AI12-0112-1} function Has_Element (Position : Cursor) return Boolean
   with Nonblocking, Global => in all, Use_Formal => null;
```
18.3/5

{*AI05-0212-1*} Returns True if Position designates an element, and returns False otherwise. 18.4/3

To be honest: {*AI05-0005-1*} {*AI05-0212-1*} This function might not detect cursors that designate deleted elements; such cursors are invalid (see below) and the result of calling Has_Element with an invalid cursor is unspecified (but not erroneous). 18.c/3

```
{AI12-0112-1} function Has_Element (Container : Set; Position : Cursor)
   return Boolean
   with Nonblocking, Global => null, Use_Formal => null;
```
18.5/5

{*AI12-0112-1*} Returns True if Position designates an element in Container, and returns False otherwise. 18.6/5

Ramification: {*AI12-0112-1*} If Position is No_Element, Has_Element returns False. 18.d/5

```
function "=" (Left, Right : Set) return Boolean;
```
19/2

{*AI95-00302-03*} If Left and Right denote the same set object, then the function returns True. If Left and Right have different lengths, then the function returns False. Otherwise, for each element *E* in Left, the function returns False if an element equal to *E* (using the generic formal equality operator) is not present in Right. If the function has not returned a result after checking all of the elements, it returns True. Any exception raised during evaluation of element equality is propagated. 20/2

Implementation Note: This wording describes the canonical semantics. However, the order and number of calls on the formal equality function is unspecified for all of the operations that use it in this package, so an implementation can call it as many or as few times as it needs to get the correct answer. Specifically, there is no requirement to call the formal equality additional times once the answer has been determined. 20.a/2

```
function Equivalent_Sets (Left, Right : Set) return Boolean;
```
21/2

{*AI95-00302-03*} If Left and Right denote the same set object, then the function returns True. If Left and Right have different lengths, then the function returns False. Otherwise, for each element *E* in Left, the function returns False if an element equivalent to *E* is not present in Right. If the function has not returned a result after checking all of the elements, it returns True. Any exception raised during evaluation of element equivalence is propagated. 22/2

22.1/5
```
function Tampering_With_Cursors_Prohibited
   (Container : Set) return Boolean
   with Nonblocking, Global => null, Use_Formal => null;
```

22.2/5 {*AI12-0112-1*} Returns True if tampering with cursors is currently prohibited for Container, and returns False otherwise.

22.a/5 **Implementation Note:** {*AI12-0112-1*} Various contracts elsewhere in this specification require that this function be implemented with synchronized data. Moreover, it is possible for tampering to be prohibited by multiple operations (sequentially or in parallel). Therefore, tampering needs to be implemented with an atomic or protected counter. The counter is initialized to zero, and is incremented when tampering is prohibited, and decremented when leaving an area that prohibited tampering. Function Tampering_With_Cursors_Prohibited returns True if the counter is nonzero. (Note that any case where the result is not well-defined for one task is incorrect use of shared variables and would be erroneous by the rules of 9.10, so no special protection is needed to read the counter.)

23/5
```
{AI12-0112-1} function To_Set (New_Item : Element_Type) return Set
   with Post => Length (To_Set'Result) = 1 and then
                not Tampering_with_Cursors_Prohibited (To_Set'Result);
```

24/2 {*AI95-00302-03*} Returns a set containing the single element New_Item.

25/5
```
{AI12-0112-1} function Length (Container : Set) return Count_Type
   with Nonblocking, Global => null, Use_Formal => null;
```

26/2 {*AI95-00302-03*} Returns the number of elements in Container.

27/5
```
function Is_Empty (Container : Set) return Boolean
   with Nonblocking, Global => null, Use_Formal => null,
        Post => Is_Empty'Result = (Length (Container) = 0);
```

28/5 {*AI95-00302-03*} {*AI12-0112-1*} Returns True if Container is empty.

29/5
```
{AI12-0112-1} procedure Clear (Container : in out Set)
   with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error,
        Post => Length (Container) = 0;
```

30/2 {*AI95-00302-03*} Removes all the elements from Container.

31/5
```
function Element (Position : Cursor) return Element_Type
   with Pre  => Position /= No_Element or else raise Constraint_Error,
        Nonblocking, Global => in all, Use_Formal => Element_Type;
```

32/5 {*AI95-00302-03*} {*AI12-0112-1*} Element returns the element designated by Position.

32.1/5
```
function Element (Container : Set;
                  Position  : Cursor) return Element_Type
   with Pre  => (Position /= No_Element
                    or else raise Constraint_Error) and then
                (Has_Element (Container, Position)
                   or else raise Program_Error),
        Nonblocking, Global => null, Use_Formal => Element_Type;
```

32.2/5 {*AI12-0112-1*} Element returns the element designated by Position.

33/5
```
procedure Replace_Element (Container : in out Set;
                           Position  : in     Cursor;
                           New_item  : in     Element_Type)
   with Pre  => (not Tampering_With_Elements_Prohibited (Container)
                    or else raise Program_Error) and then
                (Position /= No_Element
                   or else raise Constraint_Error) and then
                (Has_Element (Container, Position)
                   or else raise Program_Error);
```

34/5 {*AI95-00302-03*} {*AI12-0112-1*} {*AI12-0196-1*} Replace_Element assigns New_Item to the element designated by Position. Any exception raised by the assignment is propagated. For the

purposes of determining whether the parameters overlap in a call to Replace_Element, the Container parameter is not considered to overlap with any object [(including itself)].

Implementation Note: The final assignment may require that the node of the element be moved in the Set's data structures. That could mean that implementing this operation exactly as worded above could require the overhead of searching twice. Implementations are encouraged to avoid this extra overhead when possible, by prechecking if the old element is equivalent to the new one, by inserting a placeholder node while checking for an equivalent element, and similar optimizations. 34.a/2

The cursor still designates the same element after this operation; only the value of that element has changed. Cursors cannot include information about the relative position of an element in a Set (as they must survive insertions and deletions of other elements), so this should not pose an implementation hardship. 34.b/2

```
procedure Query_Element                                                    35/5
  (Position : in Cursor;
   Process  : not null access procedure (Element : in Element_Type))
   with Pre  => Position /= No_Element
                   or else raise Constraint_Error,
        Global => in all;
```

{*AI95-00302-03*} {*AI05-0021-1*} {*AI05-0265-1*} {*AI12-0112-1*} Query_Element calls Process.**all** with the element designated by Position as the argument. Tampering with the elements of the set that contains the element designated by Position is prohibited during the execution of the call on Process.**all**. Any exception raised by Process.**all** is propagated. 36/5

```
procedure Query_Element                                                    36.1/5
  (Container : in Set;
   Position  : in Cursor;
   Process  : not null access procedure (Element : in Element_Type))
   with Pre  => (Position /= No_Element
                   or else raise Constraint_Error) and then
                (Has_Element (Container, Position)
                   or else raise Program_Error);
```

{*AI12-0112-1*} Query_Element calls Process.**all** with the key and element from the node designated by Position as the arguments. Tampering with the elements of Container is prohibited during the execution of the call on Process.**all**. Any exception raised by Process.**all** is propagated. 36.2/5

```
{AI12-0112-1} type Constant_Reference_Type                                 36.3/5
      (Element : not null access constant Element_Type) is private
   with Implicit_Dereference => Element,
        Nonblocking, Global => in out synchronized,
        Default_Initial_Condition => (raise Program_Error);
```

{*AI05-0212-1*} The type Constant_Reference_Type needs finalization. 36.4/3

This paragraph was deleted.{*AI12-0112-1*} 36.5/5

Reason: It is expected that Constant_Reference_Type will be a controlled type, for which finalization will have some action to terminate the tampering check for the associated container. If the object is created by default, however, there is no associated container. Since this is useless, and supporting this case would take extra work, we define it to raise an exception. 36.a/3

```
{AI12-0112-1} function Constant_Reference (Container : aliased in Set;    36.6/5
                                           Position  : in Cursor)
   return Constant_Reference_Type
   with Pre  => (Position /= No_Element
                   or else raise Constraint_Error) and then
                (Has_Element (Container, Position)
                   or else raise Program_Error),
        Post => Tampering_With_Cursors_Prohibited (Container),
        Nonblocking, Global => null, Use_Formal => null;
```

{*AI05-0212-1*} {*AI05-0269-1*} This function (combined with the Constant_Indexing and Implicit_Dereference aspects) provides a convenient way to gain read access to an individual element of a set given a cursor. 36.7/3

36.8/5 {*AI05-0212-1*} {*AI05-0265-1*} {*AI12-0112-1*} Constant_Reference returns an object whose discriminant is an access value that designates the element designated by Position. Tampering with the cursors of Container is prohibited while the object returned by Constant_Reference exists and has not been finalized.

36.9/5
```
{AI12-0112-1} procedure Assign (Target : in out Set; Source : in Set)
   with Pre  => not Tampering_With_Cursors_Prohibited (Target)
                   or else raise Program_Error,
        Post => Length (Source) = Length (Target);
```

36.10/3 {*AI05-0001-1*} {*AI05-0248-1*} If Target denotes the same object as Source, the operation has no effect. Otherwise, the elements of Source are copied to Target as for an assignment_statement assigning Source to Target.

36.b/3 **Discussion:** {*AI05-0005-1*} This routine exists for compatibility with the bounded set containers. For an unbounded set, `Assign(A, B)` and `A := B` behave identically. For a bounded set, := will raise an exception if the container capacities are different, while Assign will not raise an exception if there is enough room in the target.

37/5
```
{AI12-0112-1} procedure Move (Target : in out Set;
                              Source : in out Set)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Target)
                    or else raise Program_Error) and then
                (not Tampering_With_Cursors_Prohibited (Source)
                    or else raise Program_Error),
        Post => (if not Target'Has_Same_Storage (Source) then
                    Length (Target) = Length (Source'Old) and then
                    Length (Source) = 0);
```

38/3 {*AI95-00302-03*} {*AI05-0001-1*} {*AI05-0248-1*} {*AI05-0262-1*} If Target denotes the same object as Source, then the operation has no effect. Otherwise, the operation is equivalent to Assign (Target, Source) followed by Clear (Source).

39/5
```
{AI12-0112-1} procedure Insert (Container : in out Set;
                                New_Item  : in     Element_Type;
                                Position  :    out Cursor;
                                Inserted  :    out Boolean)
   with Pre  => (not Tampering_With_Elements_Prohibited (Container)
                    or else raise Program_Error) and then
                (Length (Container) <= Count_Type'Last - 1
                    or else raise Constraint_Error),
        Post => (declare
                    Original_Length : constant Count_Type :=
                       Length (Container)'Old;
                 begin
                    Has_Element (Container, Position) and then
                   (if Inserted then
                      Length (Container) = Original_Length + 1
                    else
                      Length (Container) = Original_Length));
```

40/2 {*AI95-00302-03*} Insert checks if an element equivalent to New_Item is already present in Container. If a match is found, Inserted is set to False and Position designates the matching element. Otherwise, Insert adds New_Item to Container; Inserted is set to True and Position designates the newly-inserted element. Any exception raised during allocation is propagated and Container is not modified.

41/5
```
{AI12-0112-1} procedure Insert (Container : in out Set;
                    New_Item  : in     Element_Type)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                     or else raise Program_Error) and then
                (Length (Container) <= Count_Type'Last - 1
                     or else raise Constraint_Error),
        Post => Length (Container) = Length (Container)'Old + 1;
```

42/2
{*AI95-00302-03*} Insert inserts New_Item into Container as per the four-parameter Insert, with the difference that if an element equivalent to New_Item is already in the set, then Constraint_Error is propagated.

42.a/2
Discussion: This is equivalent to:

42.b/2
```
declare
  Inserted : Boolean; C : Cursor;
begin
  Insert (Container, New_Item, C, Inserted);
  if not Inserted then
     raise Constraint_Error;
  end if;
end;
```

42.c/2
but doesn't require the hassle of **out** parameters.

43/5
```
{AI12-0112-1} procedure Include (Container : in out Set;
                    New_Item  : in     Element_Type)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                     or else raise Program_Error) and then
                (Length (Container) <= Count_Type'Last - 1
                     or else raise Constraint_Error),
        Post => (declare
                   Original_Length : constant Count_Type :=
                      Length (Container)'Old;
                 begin
                   Length (Container)
                      in Original_Length | Original_Length + 1);
```

44/2
{*AI95-00302-03*} Include inserts New_Item into Container as per the four-parameter Insert, with the difference that if an element equivalent to New_Item is already in the set, then it is replaced. Any exception raised during assignment is propagated.

45/5
```
{AI12-0112-1} procedure Replace (Container : in out Set;
                    New_Item  : in     Element_Type)
   with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                     or else raise Program_Error,
        Post => Length (Container) = Length (Container)'Old;
```

46/2
{*AI95-00302-03*} Replace checks if an element equivalent to New_Item is already in the set. If a match is found, that element is replaced with New_Item; otherwise, Constraint_Error is propagated.

47/5
```
{AI12-0112-1} procedure Exclude (Container : in out Set;
                    Item       : in     Element_Type)
   with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                     or else raise Program_Error,
        Post => (declare
                   Original_Length : constant Count_Type :=
                      Length (Container)'Old;
                 begin
                   Length (Container) in
                      Original_Length - 1 | Original_Length);
```

48/2
{*AI95-00302-03*} Exclude checks if an element equivalent to Item is present in Container. If a match is found, Exclude removes the element from the set.

49/5
```
{AI12-0112-1} procedure Delete (Container : in out Set;
                                Item      : in     Element_Type)
   with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error,
        Post => Length (Container) = Length (Container)'Old - 1;
```

50/2 {*AI95-00302-03*} Delete checks if an element equivalent to Item is present in Container. If a match is found, Delete removes the element from the set; otherwise, Constraint_Error is propagated.

51/5
```
procedure Delete (Container : in out Set;
                  Position  : in out Cursor)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error) and then
                (Position /= No_Element
                    or else raise Constraint_Error) and then
                (Has_Element (Container, Position)
                   or else raise Program_Error),
        Post => Length (Container) = Length (Container)'Old - 1 and then
                Position = No_Element;
```

52/5 {*AI95-00302-03*} {*AI12-0112-1*} Delete removes the element designated by Position from the set.

52.a/2 **Ramification:** The check on Position checks that the cursor does not belong to some other set. This check implies that a reference to the set is included in the cursor value. This wording is not meant to require detection of dangling cursors; such cursors are defined to be invalid, which means that execution is erroneous, and any result is allowed (including not raising an exception).

53/5
```
{AI12-0112-1} procedure Union (Target : in out Set;
                               Source : in     Set)
   with Pre  => not Tampering_With_Cursors_Prohibited (Target)
                   or else raise Program_Error,
        Post => Length (Target) <= Length (Target)'Old + Length (Source);
```

54/2 {*AI95-00302-03*} Union inserts into Target the elements of Source that are not equivalent to some element already in Target.

54.a/2 **Implementation Note:** If the objects are the same, the result is the same as the original object. The implementation needs to take care so that aliasing effects do not make the result trash; Union (S, S); must work.

55/5
```
{AI12-0112-1} function Union (Left, Right : Set) return Set
   with Post => Length (Union'Result) <=
                   Length (Left) + Length (Right) and then
                not Tampering_With_Cursors_Prohibited (Union'Result);
```

56/2 {*AI95-00302-03*} Returns a set comprising all of the elements of Left, and the elements of Right that are not equivalent to some element of Left.

57/5
```
{AI12-0112-1} procedure Intersection (Target : in out Set;
                                      Source : in     Set)
   with Pre  => not Tampering_With_Cursors_Prohibited (Target)
                   or else raise Program_Error,
        Post => Length (Target) <= Length (Target)'Old + Length (Source);
```

58/3 {*AI95-00302-03*} {*AI05-0004-1*} Intersection deletes from Target the elements of Target that are not equivalent to some element of Source.

58.a/2 **Implementation Note:** If the objects are the same, the result is the same as the original object. The implementation needs to take care so that aliasing effects do not make the result trash; Intersection (S, S); must work.

```
{AI12-0112-1} function Intersection (Left, Right : Set) return Set                    59/5
   with Post => Length (Intersection'Result) <=
                   Length (Left) + Length (Right) and then
                not Tampering_With_Cursors_Prohibited (Intersection'Result);
```

{*AI95-00302-03*} Returns a set comprising all the elements of Left that are equivalent to the some element of Right. 60/2

```
{AI12-0112-1} procedure Difference (Target : in out Set;                              61/5
                       Source : in     Set)
   with Pre  => not Tampering_With_Cursors_Prohibited (Target)
                   or else raise Program_Error,
        Post => Length (Target) <= Length (Target)'Old + Length (Source);
```

{*AI95-00302-03*} If Target denotes the same object as Source, then Difference clears Target. Otherwise, it deletes from Target the elements that are equivalent to some element of Source. 62/2

```
{AI12-0112-1} function Difference (Left, Right : Set) return Set                      63/5
   with Post => Length (Difference'Result) <= Length (Left) +
                   Length (Right) and then
                not Tampering_With_Cursors_Prohibited (Difference'Result);
```

{*AI95-00302-03*} Returns a set comprising the elements of Left that are not equivalent to some element of Right. 64/2

```
{AI12-0112-1} procedure Symmetric_Difference (Target : in out Set;                    65/5
                              Source : in     Set)
   with Pre  => not Tampering_With_Cursors_Prohibited (Target)
                   or else raise Program_Error,
        Post => Length (Target) <= Length (Target)'Old + Length (Source);
```

{*AI95-00302-03*} If Target denotes the same object as Source, then Symmetric_Difference clears Target. Otherwise, it deletes from Target the elements that are equivalent to some element of Source, and inserts into Target the elements of Source that are not equivalent to some element of Target. 66/2

```
{AI12-0112-1} function Symmetric_Difference (Left, Right : Set) return Set            67/5
   with Post => Length (Symmetric_Difference'Result) <=
                   Length (Left) + Length (Right) and then
                not Tampering_With_Cursors_Prohibited (
                   Symmetric_Difference'Result);
```

{*AI95-00302-03*} Returns a set comprising the elements of Left that are not equivalent to some element of Right, and the elements of Right that are not equivalent to some element of Left. 68/2

```
function Overlap (Left, Right : Set) return Boolean;                                  69/2
```

{*AI95-00302-03*} {*AI05-0264-1*} If an element of Left is equivalent to some element of Right, then Overlap returns True. Otherwise, it returns False. 70/3

Discussion: This operation is commutative. If Overlap returns False, the two sets are disjoint. 70.a/2

```
function Is_Subset (Subset : Set;                                                     71/2
                    Of_Set : Set) return Boolean;
```

{*AI95-00302-03*} {*AI05-0264-1*} If an element of Subset is not equivalent to some element of Of_Set, then Is_Subset returns False. Otherwise, it returns True. 72/3

Discussion: This operation is not commutative, so we use parameter names that make it clear in named notation which set is which. 72.a/2

73/5
```
{AI12-0112-1} function First (Container : Set) return Cursor
   with Nonblocking, Global => null, Use_Formal => null,
        Post => (if not Is_Empty (Container)
                 then Has_Element (Container, First'Result)
                 else First'Result = No_Element);
```

74/2 {*AI95-00302-03*} If Length (Container) = 0, then First returns No_Element. Otherwise, First returns a cursor that designates the first element in Container.

75/5
```
{AI12-0112-1} function Next (Position : Cursor) return Cursor
   with Nonblocking, Global => in all, Use_Formal => null,
        Post => (if Position = No_Element then Next'Result = No_Element);
```

76/2 {*AI95-00302-03*} Returns a cursor that designates the successor of the element designated by Position. If Position designates the last element, then No_Element is returned. If Position equals No_Element, then No_Element is returned.

76.1/5
```
function Next (Container : Set;
               Position  : Cursor) return Cursor
   with Nonblocking, Global => null, Use_Formal => null,
        Pre  => Position = No_Element or else
                Has_Element (Container, Position)
                   or else raise Program_Error,
        Post => (if Position = No_Element then Next'Result = No_Element
                 elsif Next'Result = No_Element then
                   Position = Last (Container)
                 else Has_Element (Container, Next'Result));
```

76.2/5 {*AI12-0112-1*} Returns a cursor designating the successor of the node designated by Position in Container.

77/5
```
{AI12-0112-1} procedure Next (Position : in out Cursor)
   with Nonblocking, Global => in all, Use_Formal => null;
```

78/2 {*AI95-00302-03*} Equivalent to Position := Next (Position).

78.1/5
```
procedure Next (Container : in     Set;
                Position  : in out Cursor)
   with Nonblocking, Global => null, Use_Formal => null,
        Pre  => Position = No_Element or else
                Has_Element (Container, Position)
                   or else raise Program_Error,
        Post => (if Position /= No_Element
                 then Has_Element (Container, Position));
```

78.2/5 {*AI12-0112-1*} Equivalent to Position := Next (Container, Position).

79/3 *This paragraph was deleted.*{*AI95-00302-03*} {*AI05-0004-1*}

80/5
```
{AI12-0112-1} function Find (Container : Set;
              Item      : Element_Type) return Cursor
      with Post => (if Find'Result /= No_Element
                    then Has_Element (Container, Find'Result));
```

81/2 {*AI95-00302-03*} If Length (Container) equals 0, then Find returns No_Element. Otherwise, Find checks if an element equivalent to Item is present in Container. If a match is found, a cursor designating the matching element is returned; otherwise, No_Element is returned.

82/2
```
function Contains (Container : Set;
                   Item      : Element_Type) return Boolean;
```

82.1/3 {*AI05-0004-1*} Equivalent to Find (Container, Item) /= No_Element.

Paragraphs 83 and 84 were moved above.

```
{AI12-0112-1} procedure Iterate                                                    85/5
  (Container : in Set;
   Process   : not null access procedure (Position : in Cursor))
   with Allows_Exit;
```

{*AI95-00302-03*} {*AI05-0265-1*} Iterate calls Process.**all** with a cursor that designates each element in Container, starting with the first element and moving the cursor according to the successor relation. Tampering with the cursors of Container is prohibited during the execution of a call on Process.**all**. Any exception raised by Process.**all** is propagated. 86/3

Implementation Note: The "tamper with cursors" check takes place when the operations that insert or delete elements, and so on are called. 86.a/2

See Iterate for vectors (A.18.2) for a suggested implementation of the check. 86.b/2

{*AI95-00302-03*} Both Containers.Hashed_Set and Containers.Ordered_Set declare a nested generic package Generic_Keys, which provides operations that allow set manipulation in terms of a key (typically, a portion of an element) instead of a complete element. The formal function Key of Generic_Keys extracts a key value from an element. It is expected to return the same value each time it is called with a particular element. The behavior of Generic_Keys is unspecified if Key behaves in some other manner. 87/2

{*AI95-00302-03*} A key is expected to unambiguously determine a single equivalence class for elements. The behavior of Generic_Keys is unspecified if the formal parameters of this package behave in some other manner. 88/2

```
{AI12-0112-1} function Key (Position : Cursor) return Key_Type                     89/5
   with Pre  => Position /= No_Element or else raise Constraint_Error,
        Global => in all;
```

{*AI95-00302-03*} Equivalent to Key (Element (Position)). 90/2

```
function Key (Container : Set;                                                     90.1/5
              Position : Cursor) return Key_Type
   with Pre  => (Position /= No_Element
                    or else raise Constraint_Error) and then
                 (Has_Element (Container, Position)
                    or else raise Program_Error);
```

{*AI12-0112-1*} Equivalent to Key (Element (Container, Position)). 90.2/5

{*AI95-00302-03*} The subprograms in package Generic_Keys named Contains, Find, Element, Delete, and Exclude, are equivalent to the corresponding subprograms in the parent package, with the difference that the Key parameter is used to locate an element in the set. 91/2

```
{AI12-0112-1} procedure Replace (Container : in out Set;                           92/5
                  Key       : in     Key_Type;
                  New_Item  : in     Element_Type)
   with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error,
        Post => Length (Container) = Length (Container)'Old;
```

{*AI95-00302-03*} Equivalent to Replace_Element (Container, Find (Container, Key), New_Item). 93/2

94/5
```
procedure Update_Element_Preserving_Key
   (Container : in out Set;
    Position  : in     Cursor;
    Process   : not null access procedure
                                 (Element : in out Element_Type))
   with Pre  => (Position /= No_Element
                    or else raise Constraint_Error) and then
                (Has_Element (Container, Position)
                    or else raise Program_Error);
```

95/5 {*AI95-00302-03*} {*AI05-0265-1*} {*AI12-0112-1*} Update_Element_Preserving_Key uses Key to save the key value *K* of the element designated by Position. Update_Element_Preserving_Key then calls Process.**all** with that element as the argument. Tampering with the cursors of Container is prohibited during the execution of the call on Process.**all**. Any exception raised by Process.**all** is propagated. After Process.**all** returns, Update_Element_Preserving_Key checks if *K* determines the same equivalence class as that for the new element; if not, the element is removed from the set and Program_Error is propagated.

95.a/2 **Reason:** The key check ensures that the invariants of the set are preserved by the modification. The "tampers with the elements" check prevents data loss (if Element_Type is by-copy) or erroneous execution (if element type is unconstrained and indefinite).

96/2 If Element_Type is unconstrained and definite, then the actual Element parameter of Process.**all** shall be unconstrained.

96.a/2 **Ramification:** This means that the elements cannot be directly allocated from the heap; it must be possible to change the discriminants of the element in place.

96.1/5
```
{AI12-0112-1} type Reference_Type (Element : not null access Element_Type) is
private
   with Implicit_Dereference => Element,
        Nonblocking, Global => in out synchronized,
        Default_Initial_Condition => (raise Program_Error);
```

96.2/3 {*AI05-0212-1*} The type Reference_Type needs finalization.

96.3/5 *This paragraph was deleted.*{*AI12-0112-1*}

96.4/5
```
{AI12-0112-1} function Reference_Preserving_Key (Container : aliased in out Set;
                                                 Position  : in Cursor)
   return Reference_Type
   with Pre  => (Position /= No_Element
                    or else raise Constraint_Error) and then
                (Has_Element (Container, Position)
                    or else raise Program_Error),
        Post => Tampering_With_Cursors_Prohibited (Container);
```

96.5/3 {*AI05-0212-1*} {*AI05-0269-1*} This function (combined with the Implicit_Dereference aspect) provides a convenient way to gain read and write access to an individual element of a set given a cursor.

96.6/5 {*AI05-0212-1*} {*AI05-0265-1*} {*AI12-0112-1*} Reference_Preserving_Key uses Key to save the key value *K*; then returns an object whose discriminant is an access value that designates the element designated by Position. Tampering with the cursors of Container is prohibited while the object returned by Reference_Preserving_Key exists and has not been finalized. When the object returned by Reference_Preserving_Key is finalized, a check is made if *K* determines the same equivalence class as that for the new element; if not, the element is removed from the set and Program_Error is propagated.

```
{AI12-0112-1} function Constant_Reference (Container : aliased in Set;                    96.7/5
                                           Key       : in Key_Type)
   return Constant_Reference_Type
   with Pre  => Find (Container, Key) /= No_Element
                   or else raise Constraint_Error,
        Post => Tampering_With_Cursors_Prohibited (Container);
```

{*AI05-0212-1*} {*AI05-0269-1*} This function (combined with the Implicit_Dereference aspect) provides a convenient way to gain read access to an individual element of a set given a key value. 96.8/3

Equivalent to Constant_Reference (Container, Find (Container, Key)). 96.9/3

```
{AI12-0112-1} function Reference_Preserving_Key (Container : aliased in out Set;          96.10/5
                                                 Key       : in Key_Type)
   return Reference_Type
   with Pre  => Find (Container, Key) /= No_Element
                   or else raise Constraint_Error,
        Post => Tampering_With_Cursors_Prohibited (Container);
```

{*AI05-0212-1*} {*AI05-0269-1*} This function (combined with the Implicit_Dereference aspect) provides a convenient way to gain read and write access to an individual element of a set given a key value. 96.11/3

Equivalent to Reference_Preserving_Key (Container, Find (Container, Key)). 96.12/3

{*AI12-0111-1*} The nested package Stable provides a type Stable.Set that represents a *stable* set, which is one that cannot grow and shrink. Such a set can be created by calling the Copy function, or by establishing a *stabilized view* of an ordinary set. 96.13/5

{*AI12-0111-1*} The subprograms of the set package that have a parameter or result of type Set are included in the nested package Stable with the same specification, except that the following are omitted: 96.14/5

Tampering_With_Cursors_Prohibited, Assign, Move, Insert, Include, Clear, Delete, Exclude, Replace, Replace_Element, procedures Union, Intersection, Difference, and Symmetric_Difference, (for Ordered_sets) Delete_First and Delete_Last, and (for Hashed_sets) Reserve_Capacity 96.15/5

Discussion: The Generic_Keys package is not included in the Stable package. The functions Union, Intersection, Difference, and Symmetric_Difference are included in the Stable package. 96.a.1/5

Ramification: The names Set and Cursor mean the types declared in the nested package in these subprogram specifications. 96.a.2/5

Reason: The omitted routines are those that tamper with cursors (or test that state). The model is that it is impossible to tamper with cursors of a stable view since no such operations are included. Thus tampering checks are not needed for a stable view, and we omit the operations associated with those checks. 96.a.3/5

{*AI12-0111-1*} The operations of this package are equivalent to those for ordinary sets, except that the calls to Tampering_With_Cursors_Prohibited that occur in preconditions are replaced by False, and any that occur in postconditions are replaced by True. 96.16/5

{*AI12-0111-1*} {*AI12-0439-1*} If a stable set is declared with the Base discriminant designating a pre-existing ordinary set, the stable set represents a stabilized view of the underlying ordinary set, and any operation on the stable set is reflected on the underlying ordinary set. While a stabilized view exists, any operation that tampers with cursors performed on the underlying set is prohibited. The finalization of a stable set that provides such a view removes this restriction on the underlying ordinary set [(though some other restriction can exist due to other concurrent iterations or stabilized views)]. 96.17/5

{*AI12-0111-1*} {*AI12-0438-1*} If a stable set is declared without specifying Base, the object is necessarily initialized. The initializing expression of the stable set, [typically a call on Copy], determines the Length of the set. The Length of a stable set never changes after initialization. 96.18/5

Proof: {*AI12-0438-1*} Initialization is required as the type is indefinite, see 3.3.1. 96.b/5

Bounded (Run-Time) Errors

96.19/3 {*AI05-0022-1*} {*AI05-0248-1*} It is a bounded error for the actual function associated with a generic formal subprogram, when called as part of an operation of a set package, to tamper with elements of any set parameter of the operation. Either Program_Error is raised, or the operation works as defined on the value of the set either prior to, or subsequent to, some or all of the modifications to the set.

96.20/3 {*AI05-0027-1*} It is a bounded error to call any subprogram declared in the visible part of a set package when the associated container has been finalized. If the operation takes Container as an **in out** parameter, then it raises Constraint_Error or Program_Error. Otherwise, the operation either proceeds as it would for an empty container, or it raises Constraint_Error or Program_Error.

Erroneous Execution

97/2 {*AI95-00302-03*} A Cursor value is *invalid* if any of the following have occurred since it was created:

98/2 • The set that contains the element it designates has been finalized;

98.1/3 • {*AI05-0160-1*} The set that contains the element it designates has been used as the Target of a call to Assign, or as the target of an assignment_statement;

99/2 • The set that contains the element it designates has been used as the Source or Target of a call to Move; or

100/3 • {*AI05-0160-1*} {*AI05-0262-1*} The element it designates has been removed from the set that previously contained the element.

100.a/3 **Ramification:** {*AI05-0160-1*} This can happen directly via calls to Clear, Exclude, Delete, and Update_Element_Preserving_Key, and indirectly via calls to procedures Intersection, Difference, and Symmetric_Difference.

101/2 {*AI95-00302-03*} The result of "=" or Has_Element is unspecified if these functions are called with an invalid cursor parameter. Execution is erroneous if any other subprogram declared in Containers.Hashed_Sets or Containers.Ordered_Sets is called with an invalid cursor parameter.

101.a/2 **Discussion:** The list above is intended to be exhaustive. In other cases, a cursor value continues to designate its original element. For instance, cursor values survive the insertion and deletion of other elements.

101.b/2 While it is possible to check for these cases, in many cases the overhead necessary to make the check is substantial in time or space. Implementations are encouraged to check for as many of these cases as possible and raise Program_Error if detected.

101.1/3 {*AI05-0212-1*} Execution is erroneous if the set associated with the result of a call to Reference or Constant_Reference is finalized before the result object returned by the call to Reference or Constant_Reference is finalized.

101.c/3 **Reason:** Each object of Reference_Type and Constant_Reference_Type probably contains some reference to the originating container. If that container is prematurely finalized (which is only possible via Unchecked_Deallocation, as accessibility checks prevent passing a container to Reference that will not live as long as the result), the finalization of the object of Reference_Type will try to access a nonexistent object. This is a normal case of a dangling pointer created by Unchecked_Deallocation; we have to explicitly mention it here as the pointer in question is not visible in the specification of the type. (This is the same reason we have to say this for invalid cursors.)

Implementation Requirements

102/5 {*AI95-00302-03*} {*AI12-0437-1*} No storage associated with a set object shall be lost upon assignment or scope exit.

103/3 {*AI95-00302-03*} {*AI05-0262-1*} The execution of an assignment_statement for a set shall have the effect of copying the elements from the source set object to the target set object and changing the length of the target object to that of the source object.

103.a/3 **Implementation Note:** {*AI05-0298-1*} An assignment of a Set is a “deep” copy; that is the elements are copied as well as the data structures. We say “effect of” in order to allow the implementation to avoid copying elements immediately

if it wishes. For instance, an implementation that avoided copying until one of the containers is modified would be allowed. (Note that this implementation would require care, see A.18.2 for more.)

Implementation Advice

{*AI95-00302-03*} Move should not copy elements, and should minimize copying of internal data structures. 104/2

Implementation Advice: Move for sets should not copy elements, and should minimize copying of internal data structures. 104.a/2

Implementation Note: Usually that can be accomplished simply by moving the pointer(s) to the internal data structures from the Source container to the Target container. 104.b/2

{*AI95-00302-03*} If an exception is propagated from a set operation, no storage should be lost, nor any elements removed from a set unless specified by the operation. 105/2

Implementation Advice: If an exception is propagated from a set operation, no storage should be lost, nor any elements removed from a set unless specified by the operation. 105.a/2

Reason: This is important so that programs can recover from errors. But we don't want to require heroic efforts, so we just require documentation of cases where this can't be accomplished. 105.b/2

Wording Changes from Ada 95

{*AI95-00302-03*} This description of sets is new; the extensions are documented with the specific packages. 105.c/2

Extensions to Ada 2005

{*AI05-0212-1*} Added reference support to make set containers more convenient to use. 105.d/3

Wording Changes from Ada 2005

{*AI05-0001-1*} Added procedure Assign; the extension and incompatibility is documented with the specific packages. 105.e/3

{*AI05-0001-1*} Generalized the definition of Move. Specified which elements are read/written by stream attributes. 105.f/3

{*AI05-0022-1*} **Correction:** Added a Bounded (Run-Time) Error to cover tampering by generic actual subprograms. 105.g/3

{*AI05-0027-1*} **Correction:** Added a Bounded (Run-Time) Error to cover access to finalized set containers. 105.h/3

{*AI05-0160-1*} **Correction:** Revised the definition of invalid cursors to cover missing (and new) cases. 105.i/3

{*AI05-0265-1*} **Correction:** Defined when a container prohibits tampering in order to more clearly define where the check is made and the exception raised. 105.j/3

Inconsistencies With Ada 2012

{*AI12-0111-1*} Procedures Union, Intersection, Difference, and Symmeteric_Difference are now defined to tamper with the cursors of the Target parameter. A program which attempts to use one of these operations while tampering is prohibited will raise Program_Error. However, since the operations do modify the container, the effects would have been unpredictable, so this change will likely fix bugs. 105.k/5

Extensions to Ada 2012

{*AI12-0196-1*} **Correction:** Replace_Element is now defined such that it can be used concurrently so long as it operates on different elements. This allows some container operations to be used in parallel without separate synchronization. 105.l/5

Wording Changes from Ada 2012

{*AI12-0110-1*} **Corrigendum:** Clarified that tampering checks precede all other checks made by a subprogram (but come after those associated with the call). 105.m/4

{*AI12-0112-1*} Added contracts to this container. This includes describing some of the semantics with pre- and postconditions, rather than English text. Note that the preconditions can be Suppressed (see 11.5). 105.n/5

A.18.8 The Generic Package Containers.Hashed_Sets

Static Semantics

1/2 {*AI95-00302-03*} The generic library package Containers.Hashed_Sets has the following declaration:

2/5
```
{AI05-0084-1} {AI05-0212-1} {AI12-0112-1} with Ada.Iterator_Interfaces;
generic
   type Element_Type is private;
   with function Hash (Element : Element_Type) return Hash_Type;
   with function Equivalent_Elements (Left, Right : Element_Type)
                 return Boolean;
   with function "=" (Left, Right : Element_Type) return Boolean is <>;
package Ada.Containers.Hashed_Sets
   with Preelaborate, Remote_Types,
        Nonblocking, Global => in out synchronized is
```

2.a/5 **Discussion:** {*AI12-0112-1*} For discussion on the reasons and meaning of the specifications of the Global and Nonblocking aspects in this generic package, see the notes on the equivalent operations in the specification of the Containers.Vectors package (see A.18.2).

3/5
```
{AI05-0212-1} {AI12-0111-1} {AI12-0112-1} {AI12-0212-1} {AI12-0339-1} {AI12-0399-1}
type Set is tagged private
      with Constant_Indexing => Constant_Reference,
           Default_Iterator  => Iterate,
           Iterator_Element  => Element_Type,
           Iterator_View     => Stable.Set,
           Aggregate         => (Empty       => Empty,
                                 Add_Unnamed => Include),
           Stable_Properties => (Length,
                                 Tampering_With_Cursors_Prohibited),
           Default_Initial_Condition =>
              Length (Set) = 0 and then
              (not Tampering_With_Cursors_Prohibited (Set)),
           Preelaborable_Initialization;
```

3.a/5 **Discussion:** {*AI12-0112-1*} Unlike a Vector, the Stable_Properties of a Hashed_Set do not include the Capacity. If we had included it, some of the otherwise shared definitions would need different postconditions for Hashed_Sets and Ordered_Sets. If we were starting these containers from scratch, we probably would have approached the sharing of definitions differently so that we could avoid issues like this, but a major reorganization of this existing material would be too much change.

4/5
```
{AI12-0399-1}    type Cursor is private
      with Preelaborable_Initialization;
```

5/2
```
   Empty_Set : constant Set;
```

6/2
```
   No_Element : constant Cursor;
```

6.1/5
```
{AI05-0212-1} {AI12-0112-1}    function Has_Element (Position : Cursor) return
Boolean
      with Nonblocking, Global => in all, Use_Formal => null;
```

6.2/5
```
{AI12-0112-1}    function Has_Element (Container : Set; Position : Cursor)
      return Boolean
      with Nonblocking, Global => null, Use_Formal => null;
```

6.3/3
```
{AI05-0212-1}    package Set_Iterator_Interfaces is new
       Ada.Iterator_Interfaces (Cursor, Has_Element);
```

7/2
```
   function "=" (Left, Right : Set) return Boolean;
```

8/2
```
   function Equivalent_Sets (Left, Right : Set) return Boolean;
```

8.1/5
```
{AI12-0112-1}    function Tampering_With_Cursors_Prohibited
      (Container : Set) return Boolean
      with Nonblocking, Global => null, Use_Formal => null;
```

```
{AI12-0339-1}    function Empty (Capacity : Count_Type := implementation-defined)                8.2/5
      return Set
      with Post =>
              Capacity (Empty'Result) >= Capacity and then
              not Tampering_With_Cursors_Prohibited (Empty'Result) and then
              Length (Empty'Result) = 0;

{AI12-0112-1}    function To_Set (New_Item : Element_Type) return Set                            9/5
      with Post => Length (To_Set'Result) = 1 and then
                   not Tampering_with_Cursors_Prohibited (To_Set'Result);

{AI12-0112-1}    function Capacity (Container : Set) return Count_Type                           10/5
      with Nonblocking, Global => null, Use_Formal => null;

{AI12-0112-1}    procedure Reserve_Capacity (Container : in out Set;                             11/5
                             Capacity  : in     Count_Type)
      with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                      or else raise Program_Error,
           Post => Container.Capacity >= Capacity;

{AI12-0112-1}    function Length (Container : Set) return Count_Type                             12/5
      with Nonblocking, Global => null, Use_Formal => null;

{AI12-0112-1}    function Is_Empty (Container : Set) return Boolean                              13/5
      with Nonblocking, Global => null, Use_Formal => null,
           Post => Is_Empty'Result = (Length (Container) = 0);

{AI12-0112-1}    procedure Clear (Container : in out Set)                                        14/5
      with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                      or else raise Program_Error,
           Post => Capacity (Container) = Capacity (Container)'Old and then
                   Length (Container) = 0;

{AI12-0112-1}    function Element (Position : Cursor) return Element_Type                        15/5
      with Pre  => Position /= No_Element or else raise Constraint_Error,
           Nonblocking, Global => in all, Use_Formal => Element_Type;

{AI12-0112-1}    function Element (Container : Set;                                              15.1/5
                                   Position  : Cursor) return Element_Type
      with Pre  => (Position /= No_Element
                      or else raise Constraint_Error) and then
                   (Has_Element (Container, Position)
                      or else raise Program_Error),
           Nonblocking, Global => null, Use_Formal => Element_Type;

{AI12-0112-1}    procedure Replace_Element (Container : in out Set;                              16/5
                                            Position  : in     Cursor;
                                            New_item  : in     Element_Type)
      with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                      or else raise Program_Error) and then
                   (Position /= No_Element
                      or else raise Constraint_Error) and then
                   (Has_Element (Container, Position)
                      or else raise Program_Error);

{AI12-0112-1}    procedure Query_Element                                                         17/5
     (Position : in Cursor;
      Process  : not null access procedure (Element : in Element_Type))
      with Pre  => Position /= No_Element or else raise Constraint_Error,
           Global => in all;

{AI12-0112-1}    procedure Query_Element                                                         17.1/5
     (Container : in Set;
      Position  : in Cursor;
      Process   : not null access procedure (Element : in Element_Type))
      with Pre  => (Position /= No_Element
                      or else raise Constraint_Error) and then
                   (Has_Element (Container, Position)
                      or else raise Program_Error);
```

17.2/5 {*AI05-0212-1*} {*AI12-0112-1*}
```
type Constant_Reference_Type
   (Element : not null access constant Element_Type) is private
with Implicit_Dereference => Element,
     Nonblocking, Global => in out synchronized,
     Default_Initial_Condition => (raise Program_Error);
```

17.3/5 {*AI05-0212-1*} {*AI12-0112-1*}
```
function Constant_Reference (Container : aliased in Set;
                             Position  : in Cursor)
   return Constant_Reference_Type
   with Pre  => (Position /= No_Element
                   or else raise Constraint_Error) and then
                (Has_Element (Container, Position)
                   or else raise Program_Error),
        Post => Tampering_With_Cursors_Prohibited (Container),
        Nonblocking, Global => null, Use_Formal => null;
```

17.4/5 {*AI05-0001-1*} {*AI12-0112-1*}
```
procedure Assign (Target : in out Set; Source : in Set)
   with Pre  => not Tampering_With_Cursors_Prohibited (Target)
                   or else raise Program_Error,
        Post => Length (Source) = Length (Target) and then
                Capacity (Target) >= Length (Source);
```

17.5/5 {*AI05-0001-1*} {*AI12-0112-1*}
```
function Copy (Source : Set; Capacity : Count_Type := 0)
   return Set
   with Pre  => Capacity = 0 or else Capacity >= Length (Source)
                   or else raise Capacity_Error,
        Post =>
           Length (Copy'Result) = Length (Source) and then
           not Tampering_With_Cursors_Prohibited (Copy'Result) and then
           Copy'Result.Capacity = (if Capacity = 0 then
              Length (Source) else Capacity);
```

18/5 {*AI12-0112-1*}
```
procedure Move (Target : in out Set;
                Source : in out Set)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Target)
                   or else raise Program_Error) and then
                (not Tampering_With_Cursors_Prohibited (Source)
                   or else raise Program_Error),
        Post => (if not Target'Has_Same_Storage (Source) then
                   Length (Target) = Length (Source'Old) and then
                   Length (Source) = 0);
```

19/5 {*AI12-0112-1*}
```
procedure Insert (Container : in out Set;
                  New_Item  : in     Element_Type;
                  Position  :    out Cursor;
                  Inserted  :    out Boolean)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                   or else raise Program_Error) and then
                (Length (Container) <= Count_Type'Last - 1
                   or else raise Constraint_Error),
        Post => (declare
                   Original_Length : constant Count_Type :=
                      Length (Container)'Old;
                 begin
                   Has_Element (Container, Position) and then
                   (if Inserted then
                      Length (Container) = Original_Length + 1
                    else
                      Length (Container) = Original_Length)) and then
                 Capacity (Container) >= Length (Container);
```

```
{AI12-0112-1}    procedure Insert (Container : in out Set;                    20/5
                     New_Item  : in     Element_Type)
      with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                       or else raise Program_Error) and then
                    (Length (Container) <= Count_Type'Last - 1
                       or else raise Constraint_Error),
           Post => Length (Container) = Length (Container)'Old + 1 and then
                   Capacity (Container) >= Length (Container);

{AI12-0112-1}    procedure Include (Container : in out Set;                   21/5
                     New_Item  : in     Element_Type)
      with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                       or else raise Program_Error) and then
                    (Length (Container) <= Count_Type'Last - 1
                       or else raise Constraint_Error),
           Post => (declare
                      Original_Length : constant Count_Type :=
                         Length (Container)'Old;
                    begin
                      Length (Container)
                         in Original_Length | Original_Length + 1) and then
                     Capacity (Container) >= Length (Container);

{AI12-0112-1}    procedure Replace (Container : in out Set;                   22/5
                     New_Item  : in     Element_Type)
      with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                       or else raise Program_Error,
           Post => Length (Container) = Length (Container)'Old;

{AI12-0112-1}    procedure Exclude (Container : in out Set;                   23/5
                     Item      : in     Element_Type)
      with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                       or else raise Program_Error,
           Post => (declare
                      Original_Length : constant Count_Type :=
                         Length (Container)'Old;
                    begin
                      Length (Container) in
                         Original_Length - 1 | Original_Length);

{AI12-0112-1}    procedure Delete (Container : in out Set;                    24/5
                      Item      : in     Element_Type)
      with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                       or else raise Program_Error,
           Post => Length (Container) = Length (Container)'Old - 1;

{AI12-0112-1}    procedure Delete (Container : in out Set;                    25/5
                      Position  : in out Cursor)
      with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                       or else raise Program_Error) and then
                    (Position /= No_Element
                       or else raise Constraint_Error) and then
                    (Has_Element (Container, Position)
                       or else raise Program_Error),
           Post => Length (Container) = Length (Container)'Old - 1 and then
                   Position = No_Element;

{AI12-0112-1}    procedure Union (Target : in out Set;                        26/5
                     Source : in     Set)
      with Pre  => not Tampering_With_Cursors_Prohibited (Target)
                       or else raise Program_Error,
           Post => Length (Target) <= Length (Target)'Old + Length (Source);

{AI12-0112-1}    function Union (Left, Right : Set) return Set               27/5
      with Post => Length (Union'Result) <=
                       Length (Left) + Length (Right) and then
                   not Tampering_With_Cursors_Prohibited (Union'Result);

   function "or" (Left, Right : Set) return Set renames Union;               28/2
```

```
29/5    {AI12-0112-1}    procedure Intersection (Target : in out Set;
                                                 Source : in     Set)
              with Pre  => not Tampering_With_Cursors_Prohibited (Target)
                               or else raise Program_Error,
                   Post => Length (Target) <= Length (Target)'Old + Length (Source);

30/5    {AI12-0112-1}    function Intersection (Left, Right : Set) return Set
              with Post =>
                      Length (Intersection'Result) <=
                         Length (Left) + Length (Right) and then
                      not Tampering_With_Cursors_Prohibited (Intersection'Result);

31/2       function "and" (Left, Right : Set) return Set renames Intersection;

32/5    {AI12-0112-1}    procedure Difference (Target : in out Set;
                                              Source : in     Set)
              with Pre  => not Tampering_With_Cursors_Prohibited (Target)
                               or else raise Program_Error,
                   Post => Length (Target) <= Length (Target)'Old + Length (Source);

33/5    {AI12-0112-1}    function Difference (Left, Right : Set) return Set
              with Post =>
                      Length (Difference'Result) <=
                         Length (Left) + Length (Right) and then
                      not Tampering_With_Cursors_Prohibited (Difference'Result);

34/2       function "-" (Left, Right : Set) return Set renames Difference;

35/5    {AI12-0112-1}    procedure Symmetric_Difference (Target : in out Set;
                                                        Source : in     Set)
              with Pre  => not Tampering_With_Cursors_Prohibited (Target)
                               or else raise Program_Error,
                   Post => Length (Target) <= Length (Target)'Old + Length (Source);

36/5    {AI12-0112-1}    function Symmetric_Difference (Left, Right : Set) return Set
              with Post =>
                      Length (Symmetric_Difference'Result) <=
                         Length (Left) + Length (Right) and then
                      not Tampering_With_Cursors_Prohibited (
                         Symmetric_Difference'Result);

37/2       function "xor" (Left, Right : Set) return Set
             renames Symmetric_Difference;

38/2       function Overlap (Left, Right : Set) return Boolean;

39/2       function Is_Subset (Subset : Set;
                               Of_Set : Set) return Boolean;

40/5    {AI12-0112-1}    function First (Container : Set) return Cursor
              with Nonblocking, Global => null, Use_Formal => null,
                   Post => (if not Is_Empty (Container)
                            then Has_Element (Container, First'Result)
                            else First'Result = No_Element);

41/5    {AI12-0112-1}    function Next (Position : Cursor) return Cursor
              with Nonblocking, Global => in all, Use_Formal => null,
                   Post => (if Position = No_Element then Next'Result = No_Element);

41.1/5  {AI12-0112-1}    function Next (Container : Set;
                                        Position : Cursor) return Cursor
              with Nonblocking, Global => null, Use_Formal => null,
                   Pre  => Position = No_Element or else
                           Has_Element (Container, Position)
                              or else raise Program_Error,
                   Post => (if Position = No_Element then Next'Result = No_Element
                            elsif Next'Result = No_Element then
                              Position = Last (Container)
                            else Has_Element (Container, Next'Result));

42/5    {AI12-0112-1}    procedure Next (Position : in out Cursor)
              with Nonblocking, Global => in all, Use_Formal => null;
```

```
{AI12-0112-1}    procedure Next (Container : in     Set;                                    42.1/5
                                 Position  : in out Cursor)
      with Nonblocking, Global => null, Use_Formal => null,
           Pre  => Position = No_Element or else
                   Has_Element (Container, Position)
                      or else raise Program_Error,
           Post => (if Position /= No_Element
                    then Has_Element (Container, Position));

{AI12-0112-1}    function Find (Container : Set;                                             43/5
                                Item      : Element_Type)
      return Cursor
      with Post => (if Find'Result /= No_Element
                    then Has_Element (Container, Find'Result));

   function Contains (Container : Set;                                                      44/2
                      Item      : Element_Type) return Boolean;

This paragraph was deleted.{AI05-0212-1}                                                    45/3

{AI12-0112-1}    function Equivalent_Elements (Left, Right : Cursor)                        46/5
      return Boolean
      with Pre    => (Left /= No_Element and then Right /= No_Element)
                        or else raise Constraint_Error,
           Global => in all;

{AI12-0112-1}    function Equivalent_Elements (Left  : Cursor;                              47/5
                                               Right : Element_Type)
      return Boolean
      with Pre    => Left /= No_Element or else raise Constraint_Error,
           Global => in all;

{AI12-0112-1}    function Equivalent_Elements (Left  : Element_Type;                        48/5
                                               Right : Cursor)
      return Boolean
      with Pre    => Right /= No_Element or else raise Constraint_Error,
           Global => in all;

{AI12-0112-1}    procedure Iterate                                                          49/5
     (Container : in Set;
      Process   : not null access procedure (Position : in Cursor))
      with Allows_Exit;

{AI05-0212-1} {AI12-0112-1} {AI12-0266-1}    function Iterate (Container : in Set)         49.1/5
      return Set_Iterator_Interfaces.Parallel_Iterator'Class
      with Post => Tampering_With_Cursors_Prohibited (Container);

{AI12-0112-1}    generic                                                                    50/5
      type Key_Type (<>) is private;
      with function Key (Element : Element_Type) return Key_Type;
      with function Hash (Key : Key_Type) return Hash_Type;
      with function Equivalent_Keys (Left, Right : Key_Type)
                                        return Boolean;
   package Generic_Keys
   with Nonblocking, Global => null is

{AI12-0112-1}       function Key (Position : Cursor) return Key_Type                        51/5
         with Pre  => Position /= No_Element or else raise Constraint_Error,
              Global => in all;

{AI12-0112-1}       function Key (Container : Set;                                      51.1/5
                                  Position : Cursor) return Key_Type
         with Pre  => (Position = No_Element
                           or else raise Constraint_Error) and then
                      (Has_Element (Container, Position)
                           or else raise Program_Error);

      function Element (Container : Set;                                                    52/2
                        Key       : Key_Type)
        return Element_Type;
```

```
53/5    {AI12-0112-1}       procedure Replace (Container : in out Set;
                                    Key       : in     Key_Type;
                                    New_Item  : in     Element_Type)
                    with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                                    or else raise Program_Error,
                         Post => Length (Container) = Length (Container)'Old;

54/5    {AI12-0112-1}       procedure Exclude (Container : in out Set;
                                    Key       : in     Key_Type)
                    with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                                    or else raise Program_Error,
                         Post => (declare
                                    Original_Length : constant Count_Type :=
                                       Length (Container)'Old;
                                  begin
                                    Length (Container)
                                       in Original_Length - 1 | Original_Length);

55/5    {AI12-0112-1}       procedure Delete (Container : in out Set;
                                   Key       : in     Key_Type)
                    with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                                    or else raise Program_Error,
                         Post => Length (Container) = Length (Container)'Old - 1;

56/5    {AI12-0112-1}       function Find (Container : Set;
                             Key       : Key_Type)
                    return Cursor
                    with Post => (if Find'Result = No_Element
                                  then Has_Element (Container, Find'Result));

57/2          function Contains (Container : Set;
                                 Key       : Key_Type)
                    return Boolean;

58/5    {AI12-0112-1}       procedure Update_Element_Preserving_Key
                   (Container : in out Set;
                    Position  : in     Cursor;
                    Process   : not null access procedure
                                    (Element : in out Element_Type))
                    with Pre  => (Position /= No_Element or else
                                    raise Constraint_Error) and then
                                 (Has_Element (Container, Position) or else
                                    raise Program_Error);

58.1/5  {AI05-0212-1} {AI12-0112-1}       type Reference_Type
                      (Element : not null access Element_Type) is private
                    with Implicit_Dereference => Element,
                         Nonblocking, Global => in out synchronized,
                         Default_Initial_Condition => (raise Program_Error);

58.2/5  {AI05-0212-1} {AI12-0112-1}       function Reference_Preserving_Key (Container :
aliased in out Set;
                                                 Position  : in Cursor)
                    return Reference_Type
                    with Pre  => (Position /= No_Element
                                    or else raise Constraint_Error) and then
                                 (Has_Element (Container, Position)
                                    or else raise Program_Error),
                         Post => Tampering_With_Cursors_Prohibited (Container);

58.3/5  {AI05-0212-1} {AI12-0112-1}       function Constant_Reference (Container : aliased
in Set;
                                          Key       : in Key_Type)
                    return Constant_Reference_Type
                    with Pre  => Find (Container, Key) /= No_Element
                                    or else raise Constraint_Error,
                         Post => Tampering_With_Cursors_Prohibited (Container);
```

```
{AI05-0212-1} {AI12-0112-1}       function Reference_Preserving_Key (Container :      58.4/5
aliased in out Set;
                                          Key       : in Key_Type)
         return Reference_Type
         with Pre  => Find (Container, Key) /= No_Element
                      or else raise Constraint_Error,
              Post => Tampering_With_Cursors_Prohibited (Container);

   end Generic_Keys;                                                                  59/2

{AI12-0111-1}    package Stable is                                                   59.1/5

{AI12-0111-1} {AI12-0339-1} {AI12-0399-1} {AI12-0407-1}       type Set (Base : not   59.2/5
null access Hashed_Sets.Set) is
         tagged limited private
         with Constant_Indexing => Constant_Reference,
              Default_Iterator  => Iterate,
              Iterator_Element  => Element_Type,
              Stable_Properties => (Length),
              Global            => null,
              Default_Initial_Condition => Length (Set) = 0,
              Preelaborable_Initialization;

{AI12-0111-1} {AI12-0399-1}       type Cursor is private                           59.3/5
         with Preelaborable_Initialization;

{AI12-0111-1}       Empty_Set : constant Set;                                     59.4/5

{AI12-0111-1}       No_Element : constant Cursor;                                 59.5/5

{AI12-0111-1}       function Has_Element (Position : Cursor) return Boolean       59.6/5
         with Nonblocking, Global => in all, Use_Formal => null;

{AI12-0111-1}       package Set_Iterator_Interfaces is new                        59.7/5
         Ada.Iterator_Interfaces (Cursor, Has_Element);

{AI12-0111-1}       procedure Assign (Target : in out Hashed_Sets.Set;            59.8/5
                         Source : in Set)
         with Post => Length (Source) = Length (Target);

{AI12-0111-1}       function Copy (Source : Hashed_Sets.Set) return Set           59.9/5
         with Post => Length (Copy'Result) = Length (Source);

{AI12-0111-1}       type Constant_Reference_Type                                  59.10/5
            (Element : not null access constant Element_Type) is private
         with Implicit_Dereference => Element,
              Nonblocking, Global => null, Use_Formal => null,
              Default_Initial_Condition => (raise Program_Error);

{AI12-0111-1}       -- Additional subprograms as described in the text            59.11/5
      -- are declared here.

{AI12-0111-1}    private                                                          59.12/5

{AI12-0111-1}       ... -- not specified by the language                          59.13/5

{AI12-0111-1}    end Stable;                                                      59.14/5

private                                                                          60/2

   ... -- not specified by the language                                          61/2

end Ada.Containers.Hashed_Sets;                                                  62/2
```

{*AI95-00302-03*} An object of type Set contains an expandable hash table, which is used to provide direct access to elements. The *capacity* of an object of type Set is the maximum number of elements that can be inserted into the hash table prior to it being automatically expanded. 63/2

{*AI95-00302-03*} Two elements *E1* and *E2* are defined to be *equivalent* if Equivalent_Elements (*E1*, *E2*) returns True. 64/2

{*AI95-00302-03*} The actual function for the generic formal function Hash is expected to return the same value each time it is called with a particular element value. For any two equivalent elements, the actual for Hash is expected to return the same value. If the actual for Hash behaves in some other manner, the 65/2

behavior of this package is unspecified. Which subprograms of this package call Hash, and how many times they call it, is unspecified.

66/2 {*AI95-00302-03*} The actual function for the generic formal function Equivalent_Elements is expected to return the same value each time it is called with a particular pair of Element values. It should define an equivalence relationship, that is, be reflexive, symmetric, and transitive. If the actual for Equivalent_Elements behaves in some other manner, the behavior of this package is unspecified. Which subprograms of this package call Equivalent_Elements, and how many times they call it, is unspecified.

66.1/3 {*AI05-0044-1*} If the actual function for the generic formal function "=" returns True for any pair of nonequivalent elements, then the behavior of the container function "=" is unspecified.

67/2 {*AI95-00302-03*} If the value of an element stored in a set is changed other than by an operation in this package such that at least one of Hash or Equivalent_Elements give different results, the behavior of this package is unspecified.

67.a/2 **Discussion:** See A.18.5, “The Generic Package Containers.Hashed_Maps” for a suggested implementation, and for justification of the restrictions regarding Hash and Equivalent_Elements. Note that sets only need to store elements, not key/element pairs.

68/2 {*AI95-00302-03*} Which elements are the first element and the last element of a set, and which element is the successor of a given element, are unspecified, other than the general semantics described in A.18.7.

68.1/5
```
function Empty (Capacity : Count_Type := implementation-defined)
   return Set
   with Post =>
           Capacity (Empty'Result) >= Capacity and then
           not Tampering_With_Cursors_Prohibited (Empty'Result) and then
           Length (Empty'Result) = 0;
```

68.2/2 {*AI12-0339-1*} Returns an empty set.

69/5
```
{AI12-0112-1} function Capacity (Container : Set) return Count_Type
   with Nonblocking, Global => null, Use_Formal => null;
```

70/2 {*AI95-00302-03*} Returns the capacity of Container.

71/5
```
{AI12-0112-1} procedure Reserve_Capacity (Container : in out Set;
                          Capacity  : in     Count_Type)
   with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                   or else raise Program_Error,
        Post => Container.Capacity >= Capacity;
```

72/2 {*AI95-00302-03*} Reserve_Capacity allocates a new hash table such that the length of the resulting set can become at least the value Capacity without requiring an additional call to Reserve_Capacity, and is large enough to hold the current length of Container. Reserve_Capacity then rehashes the elements in Container onto the new hash table. It replaces the old hash table with the new hash table, and then deallocates the old hash table. Any exception raised during allocation is propagated and Container is not modified.

73/5 *This paragraph was deleted.*{*AI12-0112-1*}

73.a/2 **Reason:** Reserve_Capacity tampers with the cursors, as rehashing probably will change the relationships of the elements in Container.

74/5
```
{AI12-0112-1} procedure Clear (Container : in out Set)
   with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                   or else raise Program_Error,
        Post => Capacity (Container) = Capacity (Container)'Old and then
                Length (Container) = 0;
```

75/2 {*AI95-00302-03*} In addition to the semantics described in A.18.7, Clear does not affect the capacity of Container.

```
{AI12-0112-1} procedure Assign (Target : in out Set; Source : in Set)                    75.1/5
      with Pre  => not Tampering_With_Cursors_Prohibited (Target)
                     or else raise Program_Error,
           Post => Length (Source) = Length (Target) and then
                   Capacity (Target) >= Length (Source);
```

{*AI05-0001-1*} {*AI05-0248-1*} In addition to the semantics described in A.18.7, if the length of Source is greater than the capacity of Target, Reserve_Capacity (Target, Length (Source)) is called before assigning any elements. 75.2/3

```
function Copy (Source : Set; Capacity : Count_Type := 0)                                 75.3/5
   return Set
   with Pre  => Capacity = 0 or else Capacity >= Length (Source)
                  or else raise Capacity_Error,
        Post =>
           Length (Copy'Result) = Length (Source) and then
           not Tampering_With_Cursors_Prohibited (Copy'Result) and then
           Copy'Result.Capacity = (if Capacity = 0 then
              Length (Source) else Capacity);
```

{*AI05-0001-1*} {*AI12-0112-1*} Returns a set whose elements are initialized from the elements of Source. 75.4/5

```
{AI12-0112-1} procedure Insert (Container : in out Set;                                  76/5
                  New_Item  : in     Element_Type;
                  Position  :    out Cursor;
                  Inserted  :    out Boolean)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                  or else raise Program_Error) and then
                (Length (Container) <= Count_Type'Last - 1
                  or else raise Constraint_Error),
        Post => (declare
                  Original_Length : constant Count_Type :=
                     Length (Container)'Old;
                 begin
                  Has_Element (Container, Position) and then
                  (if Inserted then
                     Length (Container) = Original_Length + 1
                   else
                     Length (Container) = Original_Length)) and then
                 Capacity (Container) >= Length (Container);
```

{*AI95-00302-03*} In addition to the semantics described in A.18.7, if Length (Container) equals Capacity (Container), then Insert first calls Reserve_Capacity to increase the capacity of Container to some larger value. 77/2

```
function First (Container : Set) return Cursor;                                          78/2
```

{*AI95-00302-03*} If Length (Container) = 0, then First returns No_Element. Otherwise, First returns a cursor that designates the first hashed element in Container. 79/2

```
{AI12-0112-1} function Equivalent_Elements (Left, Right : Cursor)                        80/5
   return Boolean
   with Pre    => (Left /= No_Element and then Right /= No_Element)
                    or else raise Constraint_Error,
        Global => in all;
```

{*AI95-00302-03*} Equivalent to Equivalent_Elements (Element (Left), Element (Right)). 81/2

```
{AI12-0112-1} function Equivalent_Elements (Left  : Cursor;                              82/5
                                            Right : Element_Type) return Boolean
   with Pre    => Left /= No_Element or else raise Constraint_Error,
        Global => in all;
```

{*AI95-00302-03*} Equivalent to Equivalent_Elements (Element (Left), Right). 83/2

84/5
```
{AI12-0112-1} function Equivalent_Elements (Left  : Element_Type;
                                            Right : Cursor) return Boolean
   with Pre    => Right /= No_Element or else raise Constraint_Error,
        Global => in all;
```

85/2 {*AI95-00302-03*} Equivalent to Equivalent_Elements (Left, Element (Right)).

85.1/5
```
{AI12-0112-1} {AI12-0266-1} function Iterate (Container : in Set)
   return Set_Iterator_Interfaces.Parallel_Iterator'Class
   with Post => Tampering_With_Cursors_Prohibited (Container);
```

85.2/5 {*AI05-0212-1*} {*AI05-0265-1*} {*AI05-0269-1*} {*AI12-0266-1*} Iterate returns an iterator object (see 5.5.1) that will generate a value for a loop parameter (see 5.5.2) designating each element in Container, starting with the first element and moving the cursor according to the successor relation when used as a forward iterator, and processing all nodes concurrently when used as a parallel iterator. Tampering with the cursors of Container is prohibited while the iterator object exists (in particular, in the sequence_of_statements of the loop_statement whose iterator_specification denotes this object). The iterator object needs finalization.

86/2 {*AI95-00302-03*} For any element *E*, the actual function for the generic formal function Generic_Keys.Hash is expected to be such that Hash (*E*) = Generic_Keys.Hash (Key (*E*)). If the actuals for Key or Generic_Keys.Hash behave in some other manner, the behavior of Generic_Keys is unspecified. Which subprograms of Generic_Keys call Generic_Keys.Hash, and how many times they call it, is unspecified.

87/2 {*AI95-00302-03*} For any two elements *E1* and *E2*, the boolean values Equivalent_Elements (*E1*, *E2*) and Equivalent_Keys (Key (*E1*), Key (*E2*)) are expected to be equal. If the actuals for Key or Equivalent_Keys behave in some other manner, the behavior of Generic_Keys is unspecified. Which subprograms of Generic_Keys call Equivalent_Keys, and how many times they call it, is unspecified.

Implementation Advice

88/2 {*AI95-00302-03*} If *N* is the length of a set, the average time complexity of the subprograms Insert, Include, Replace, Delete, Exclude, and Find that take an element parameter should be *O*(log *N*). The average time complexity of the subprograms that take a cursor parameter should be *O*(1). The average time complexity of Reserve_Capacity should be *O*(*N*).

88.a/2 **Implementation Advice:** The average time complexity of the Insert, Include, Replace, Delete, Exclude, and Find operations of Containers.Hashed_Sets that take an element parameter should be *O*(log *N*). The average time complexity of the subprograms of Containers.Hashed_Sets that take a cursor parameter should be *O*(1). The average time complexity of Containers.Hashed_Sets.Reserve_Capacity should be *O*(*N*).

88.b/2 **Implementation Note:** {*AI95-00302-03*} See A.18.5, “The Generic Package Containers.Hashed_Maps” for implementation notes regarding some of the operations of this package.

Extensions to Ada 95

88.c/2 {*AI95-00302-03*} The generic package Containers.Hashed_Sets is new.

Incompatibilities With Ada 2005

88.d/3 {*AI05-0001-1*} Subprograms Assign and Copy are added to Containers.Hashed_Sets. If an instance of Containers.Hashed_Sets is referenced in a use_clause, and an entity *E* with the same defining_identifier as a new entity in Containers.Hashed_Sets is defined in a package that is also referenced in a use_clause, the entity *E* may no longer be use-visible, resulting in errors. This should be rare and is easily fixed if it does occur.

Extensions to Ada 2005

88.e/3 {*AI05-0212-1*} Added iterator and indexing support to make hashed set containers more convenient to use.

Wording Changes from Ada 2005

{*AI05-0044-1*} **Correction:** Added wording to require the formal function be such that equal elements are also equivalent. 88.f/3

{*AI05-0084-1*} **Correction:** Added a pragma Remote_Types so that containers can be used in distributed programs. 88.g/3

Incompatibilities With Ada 2012

{*AI12-0111-1*} {*AI12-0112-1*} {*AI12-0339-1*} A number of new subprograms, types, and even a nested package were added to Containers.Hashed_Sets to better support contracts and stable views. Therefore, a use clause conflict is possible; see the introduction of Annex A for more on this topic. 88.h/5

Extensions to Ada 2012

{*AI12-0212-1*} Sets now support positional container aggregates, so aggregate syntax can be used to create Sets. 88.i/5

{*AI12-0266-1*} The iterator for the container now can return a parallel iterator which can be used to process the container in parallel. 88.j/5

Wording Changes from Ada 2012

{*AI12-0112-1*} Added contracts to this container. This includes describing some of the semantics with pre- and postconditions, rather than English text. Note that the preconditions can be Suppressed (see 11.5). 88.k/5

A.18.9 The Generic Package Containers.Ordered_Sets

Static Semantics

{*AI95-00302-03*} The generic library package Containers.Ordered_Sets has the following declaration: 1/2

```
{AI05-0084-1} {AI05-0212-1} {AI12-0112-1} with Ada.Iterator_Interfaces;
generic
   type Element_Type is private;
   with function "<" (Left, Right : Element_Type) return Boolean is <>;
   with function "=" (Left, Right : Element_Type) return Boolean is <>;
package Ada.Containers.Ordered_Sets
   with Preelaborate, Remote_Types,
        Nonblocking, Global => in out synchronized is
```

2/5

Discussion: {*AI12-0112-1*} For discussion on the reasons and meaning of the specifications of the Global and Nonblocking aspects in this generic package, see the notes on the equivalent operations in the specification of the Containers.Vectors package (see A.18.2). 2.a/5

```
   function Equivalent_Elements (Left, Right : Element_Type) return Boolean;
```

3/2

```
{AI05-0212-1} {AI12-0111-1} {AI12-0112-1} {AI12-0212-1} {AI12-0339-1} {AI12-0399-1}
type Set is tagged private
      with Constant_Indexing => Constant_Reference,
           Default_Iterator  => Iterate,
           Iterator_Element  => Element_Type,
           Iterator_View     => Stable.Set,
           Aggregate         => (Empty       => Empty,
                                 Add_Unnamed => Include),
           Stable_Properties => (Length,
                                 Tampering_With_Cursors_Prohibited),
           Default_Initial_Condition =>
              Length (Set) = 0 and then
              (not Tampering_With_Cursors_Prohibited (Set)),
           Preelaborable_Initialization;
```

4/5

```
{AI12-0399-1}    type Cursor is private
      with Preelaborable_Initialization;
```

5/5

```
   Empty_Set : constant Set;
```

6/2

```
   No_Element : constant Cursor;
```

7/2

```
{AI05-0212-1} {AI12-0112-1}    function Has_Element (Position : Cursor) return
Boolean
      with Nonblocking, Global => in all, Use_Formal => null;
```

7.1/5

```
7.2/5       {AI12-0112-1}    function Has_Element (Container : Set; Position : Cursor)
                 return Boolean
                 with Nonblocking, Global => null, Use_Formal => null;

7.3/3       {AI05-0212-1}    package Set_Iterator_Interfaces is new
                  Ada.Iterator_Interfaces (Cursor, Has_Element);

8/2            function "=" (Left, Right : Set) return Boolean;

9/2            function Equivalent_Sets (Left, Right : Set) return Boolean;

9.1/5       {AI12-0112-1}    function Tampering_With_Cursors_Prohibited
                 (Container : Set) return Boolean
                 with Nonblocking, Global => null, Use_Formal => null;

9.2/5       {AI12-0339-1}    function Empty return Set
                 is (Empty_Set)
                 with Post =>
                         not Tampering_With_Cursors_Prohibited (Empty'Result) and then
                         Length (Empty'Result) = 0;

10/5        {AI12-0112-1}    function To_Set (New_Item : Element_Type) return Set
                 with Post => Length (To_Set'Result) = 1 and then
                              not Tampering_with_Cursors_Prohibited (To_Set'Result);

11/5        {AI12-0112-1}    function Length (Container : Set) return Count_Type
                 with Nonblocking, Global => null, Use_Formal => null;

12/5        {AI12-0112-1}    function Is_Empty (Container : Set) return Boolean
                 with Nonblocking, Global => null, Use_Formal => null,
                      Post => Is_Empty'Result = (Length (Container) = 0);

13/5        {AI12-0112-1}    procedure Clear (Container : in out Set)
                 with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                                  or else raise Program_Error,
                      Post => Length (Container) = 0;

14/5        {AI12-0112-1}    function Element (Position : Cursor) return Element_Type
                 with Pre  => Position /= No_Element or else raise Constraint_Error,
                      Nonblocking, Global => in all, Use_Formal => Element_Type;

14.1/5      {AI12-0112-1}    function Element (Container : Set;
                                   Position  : Cursor) return Element_Type
                 with Pre  => (Position /= No_Element
                                  or else raise Constraint_Error) and then
                               (Has_Element (Container, Position)
                                  or else raise Program_Error),
                      Nonblocking, Global => null, Use_Formal => Element_Type;

15/5        {AI12-0112-1}    procedure Replace_Element (Container : in out Set;
                                            Position  : in     Cursor;
                                            New_item  : in     Element_Type)
                 with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                                  or else raise Program_Error) and then
                               (Position /= No_Element
                                  or else raise Constraint_Error) and then
                               (Has_Element (Container, Position)
                                  or else raise Program_Error);

16/5        {AI12-0112-1}    procedure Query_Element
                (Position : in Cursor;
                 Process  : not null access procedure (Element : in Element_Type))
                 with Pre  => Position /= No_Element
                                  or else raise Constraint_Error,
                      Global => in all;

16.1/5      {AI12-0112-1}    procedure Query_Element
                (Container : in Set;
                 Position  : in Cursor;
                 Process   : not null access procedure (Element : in Element_Type))
                 with Pre  => (Position /= No_Element
                                  or else raise Constraint_Error) and then
                               (Has_Element (Container, Position)
                                  or else raise Program_Error);
```

```
{AI05-0212-1} {AI12-0112-1}    type Constant_Reference_Type                                    16.2/5
         (Element : not null access constant Element_Type) is private
      with Implicit_Dereference => Element,
           Nonblocking, Global => in out synchronized,
           Default_Initial_Condition => (raise Program_Error);

{AI05-0212-1} {AI12-0112-1}    function Constant_Reference (Container : aliased in            16.3/5
Set;
                                Position  : in Cursor)
      return Constant_Reference_Type
      with Pre  => (Position /= No_Element
                      or else raise Constraint_Error) and then
                   (Has_Element (Container, Position)
                      or else raise Program_Error),
           Post => Tampering_With_Cursors_Prohibited (Container),
           Nonblocking, Global => null, Use_Formal => null;

{AI05-0001-1} {AI12-0112-1}    procedure Assign (Target : in out Set; Source : in             16.4/5
Set)
      with Pre  => not Tampering_With_Cursors_Prohibited (Target)
                      or else raise Program_Error,
           Post => Length (Source) = Length (Target);

{AI05-0001-1} {AI12-0112-1}    function Copy (Source : Set) return Set                        16.5/5
      with Post => Length (Copy'Result) = Length (Source) and then
                   not Tampering_With_Cursors_Prohibited (Copy'Result);

{AI12-0112-1}    procedure Move (Target : in out Set;                                         17/5
                   Source : in out Set)
      with Pre  => (not Tampering_With_Cursors_Prohibited (Target)
                      or else raise Program_Error) and then
                   (not Tampering_With_Cursors_Prohibited (Source)
                      or else raise Program_Error),
           Post => (if not Target'Has_Same_Storage (Source) then
                      Length (Target) = Length (Source'Old) and then
                      Length (Source) = 0);

{AI12-0112-1}    procedure Insert (Container : in out Set;                                    18/5
                       New_Item  : in     Element_Type;
                       Position  :    out Cursor;
                       Inserted  :    out Boolean)
      with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                      or else raise Program_Error) and then
                   (Length (Container) <= Count_Type'Last - 1
                      or else raise Constraint_Error),
           Post => (declare
                      Original_Length : constant Count_Type :=
                         Length (Container)'Old;
                    begin
                      Has_Element (Container, Position) and then
                     (if Inserted then
                        Length (Container) = Original_Length + 1
                      else
                        Length (Container) = Original_Length));

{AI12-0112-1}    procedure Insert (Container : in out Set;                                    19/5
                       New_Item  : in     Element_Type)
      with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                      or else raise Program_Error) and then
                   (Length (Container) <= Count_Type'Last - 1
                      or else raise Constraint_Error),
           Post => Length (Container) = Length (Container)'Old + 1;
```

20/5 {*AI12-0112-1*}
```
   procedure Include (Container : in out Set;
                      New_Item  : in     Element_Type)
      with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                      or else raise Program_Error) and then
                   (Length (Container) <= Count_Type'Last - 1
                      or else raise Constraint_Error),
           Post => (declare
                      Original_Length : constant Count_Type :=
                         Length (Container)'Old;
                    begin
                      Length (Container)
                         in Original_Length | Original_Length + 1);
```

21/5 {*AI12-0112-1*}
```
   procedure Replace (Container : in out Set;
                      New_Item  : in     Element_Type)
      with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                      or else raise Program_Error,
           Post => Length (Container) = Length (Container)'Old;
```

22/5 {*AI12-0112-1*}
```
   procedure Exclude (Container : in out Set;
                      Item      : in     Element_Type)
      with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                      or else raise Program_Error,
           Post => (declare
                      Original_Length : constant Count_Type :=
                         Length (Container)'Old;
                    begin
                      Length (Container)
                         in Original_Length - 1 | Original_Length);
```

23/5 {*AI12-0112-1*}
```
   procedure Delete (Container : in out Set;
                     Item      : in     Element_Type)
      with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                      or else raise Program_Error,
           Post => Length (Container) = Length (Container)'Old - 1;
```

24/5 {*AI12-0112-1*}
```
   procedure Delete (Container : in out Set;
                     Position  : in out Cursor)
      with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                      or else raise Program_Error) and then
                   (Position /= No_Element
                      or else raise Constraint_Error) and then
                   (Has_Element (Container, Position)
                      or else raise Program_Error),
           Post => Length (Container) = Length (Container)'Old - 1 and then
                   Position = No_Element;
```

25/5 {*AI12-0112-1*}
```
   procedure Delete_First (Container : in out Set)
      with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                      or else raise Program_Error,
           Post => (declare
                      Original_Length : constant Count_Type :=
                         Length (Container)'Old;
                    begin
                      (if Original_Length = 0 then Length (Container) = 0
                       else Length (Container) = Original_Length - 1));
```

26/5 {*AI12-0112-1*}
```
   procedure Delete_Last (Container : in out Set)
      with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                      or else raise Program_Error,
           Post => (declare
                      Original_Length : constant Count_Type :=
                         Length (Container)'Old;
                    begin
                      (if Original_Length = 0 then Length (Container) = 0
                       else Length (Container) = Original_Length - 1));
```

```
{AI12-0112-1}    procedure Union (Target : in out Set;                                    27/5
                                 Source : in     Set)
      with Pre  => not Tampering_With_Cursors_Prohibited (Target)
                       or else raise Program_Error,
           Post => Length (Target) <= Length (Target)'Old + Length (Source);

{AI12-0112-1}    function Union (Left, Right : Set) return Set                           28/5
      with Post => Length (Union'Result) <=
                      Length (Left) + Length (Right) and then
                    not Tampering_With_Cursors_Prohibited (Union'Result);

   function "or" (Left, Right : Set) return Set renames Union;                           29/2

{AI12-0112-1}    procedure Intersection (Target : in out Set;                             30/5
                                        Source : in     Set)
      with Pre  => not Tampering_With_Cursors_Prohibited (Target)
                       or else raise Program_Error,
           Post => Length (Target) <= Length (Target)'Old + Length (Source);

{AI12-0112-1}    function Intersection (Left, Right : Set) return Set                    31/5
      with Post =>
             Length (Intersection'Result) <=
                Length (Left) + Length (Right) and then
             not Tampering_With_Cursors_Prohibited (Intersection'Result);

   function "and" (Left, Right : Set) return Set renames Intersection;                   32/2

{AI12-0112-1}    procedure Difference (Target : in out Set;                              33/5
                                      Source : in     Set)
      with Pre  => not Tampering_With_Cursors_Prohibited (Target)
                       or else raise Program_Error,
           Post => Length (Target) <= Length (Target)'Old + Length (Source);

{AI12-0112-1}    function Difference (Left, Right : Set) return Set                      34/5
      with Post =>
             Length (Difference'Result) <=
                Length (Left) + Length (Right) and then
             not Tampering_With_Cursors_Prohibited (Difference'Result);

   function "-" (Left, Right : Set) return Set renames Difference;                       35/2

{AI12-0112-1}    procedure Symmetric_Difference (Target : in out Set;                    36/5
                                                 Source : in     Set)
      with Pre  => not Tampering_With_Cursors_Prohibited (Target)
                       or else raise Program_Error,
           Post => Length (Target) <= Length (Target)'Old + Length (Source);

{AI12-0112-1}    function Symmetric_Difference (Left, Right : Set) return Set            37/5
      with Post =>
             Length (Symmetric_Difference'Result) <=
                Length (Left) + Length (Right) and then
             not Tampering_With_Cursors_Prohibited (
                Symmetric_Difference'Result);

   function "xor" (Left, Right : Set) return Set renames                                 38/2
      Symmetric_Difference;

   function Overlap (Left, Right : Set) return Boolean;                                  39/2

   function Is_Subset (Subset : Set;                                                     40/2
                       Of_Set : Set) return Boolean;

{AI12-0112-1}    function First (Container : Set) return Cursor                          41/5
      with Nonblocking, Global => null, Use_Formal => null,
           Post => (if not Is_Empty (Container)
                    then Has_Element (Container, First'Result)
                    else First'Result = No_Element);

{AI12-0112-1}    function First_Element (Container : Set)                                42/5
      return Element_Type
      with Pre => (not Is_Empty (Container)
                       or else raise Constraint_Error);
```

```
43/5    {AI12-0112-1}    function Last (Container : Set) return Cursor
        with Nonblocking, Global => null, Use_Formal => null,
             Post => (if not Is_Empty (Container) then
                         Has_Element (Container, Last'Result)
                      else Last'Result = No_Element);

44/5    {AI12-0112-1}    function Last_Element (Container : Set)
        return Element_Type
        with Pre => (not Is_Empty (Container)
                        or else raise Constraint_Error);

45/5    {AI12-0112-1}    function Next (Position : Cursor) return Cursor
        with Nonblocking, Global => in all, Use_Formal => null,
             Post => (if Position = No_Element then Next'Result = No_Element);

45.1/5  {AI12-0112-1}    function Next (Container : Set;
                          Position : Cursor) return Cursor
        with Nonblocking, Global => null, Use_Formal => null,
             Pre  => Position = No_Element or else
                     Has_Element (Container, Position)
                        or else raise Program_Error,
             Post => (if Position = No_Element then Next'Result = No_Element
                      elsif Next'Result = No_Element then
                         Position = Last (Container)
                      else Has_Element (Container, Next'Result));

46/5    {AI12-0112-1}    procedure Next (Position : in out Cursor)
        with Nonblocking, Global => in all, Use_Formal => null;

46.1/5  {AI12-0112-1}    procedure Next (Container : in     Set;
                            Position  : in out Cursor)
        with Nonblocking, Global => null, Use_Formal => null,
             Pre  => Position = No_Element or else
                     Has_Element (Container, Position)
                        or else raise Program_Error,
             Post => (if Position /= No_Element
                      then Has_Element (Container, Position));

47/5    {AI12-0112-1}    function Previous (Position : Cursor) return Cursor
        with Nonblocking, Global => in all, Use_Formal => null,
             Post => (if Position = No_Element then
                         Previous'Result = No_Element);

47.1/5  {AI12-0112-1}    function Previous (Container : Set;
                             Position  : Cursor) return Cursor
        with Nonblocking, Global => null, Use_Formal => null,
             Pre  => Position = No_Element or else
                     Has_Element (Container, Position)
                        or else raise Program_Error,
             Post => (if Position = No_Element then
                         Previous'Result = No_Element
                      elsif Previous'Result = No_Element then
                         Position = First (Container)
                      else Has_Element (Container, Previous'Result));

48/5    {AI12-0112-1}    procedure Previous (Position : in out Cursor)
        with Nonblocking, Global => in all,
             Use_Formal => null;

48.1/5  {AI12-0112-1}    procedure Previous (Container : in     Set;
                             Position  : in out Cursor)
        with Nonblocking, Global => null, Use_Formal => null,
             Pre  => Position = No_Element or else
                     Has_Element (Container, Position)
                        or else raise Program_Error,
             Post => (if Position /= No_Element
                      then Has_Element (Container, Position));

49/5    {AI12-0112-1}    function Find (Container : Set;
                         Item      : Element_Type) return Cursor
        with Post => (if Find'Result /= No_Element
                      then Has_Element (Container, Find'Result));
```

```
   function Floor (Container : Set;                                                    50/2
                   Item      : Element_Type) return Cursor
      with Post => (if Floor'Result /= No_Element
                    then Has_Element (Container, Floor'Result));

   function Ceiling (Container : Set;                                                  51/2
                     Item      : Element_Type) return Cursor
      with Post => (if Ceiling'Result /= No_Element
                    then Has_Element (Container, Ceiling'Result));

   function Contains (Container : Set;                                                 52/2
                      Item      : Element_Type) return Boolean;

This paragraph was deleted.{AI05-0212-1}                                              53/3

{AI12-0112-1}    function "<" (Left, Right : Cursor) return Boolean                   54/5
      with Pre    => (Left /= No_Element and then Right /= No_Element)
                        or else raise Constraint_Error,
           Global => in all;

{AI12-0112-1}    function ">" (Left, Right : Cursor) return Boolean                   55/5
      with Pre    => (Left /= No_Element and then Right /= No_Element)
                        or else raise Constraint_Error,
           Global => in all;

{AI12-0112-1}    function "<" (Left : Cursor; Right : Element_Type) return Boolean    56/5
      with Pre    => Left /= No_Element or else raise Constraint_Error,
           Global => in all;

{AI12-0112-1}    function ">" (Left : Cursor; Right : Element_Type) return Boolean    57/5
      with Pre    => Left /= No_Element or else raise Constraint_Error,
           Global => in all;

{AI12-0112-1}    function "<" (Left : Element_Type; Right : Cursor) return Boolean    58/5
      with Pre    => Right /= No_Element or else raise Constraint_Error,
           Global => in all;

{AI12-0112-1}    function ">" (Left : Element_Type; Right : Cursor) return Boolean    59/5
      with Pre    => Right /= No_Element or else raise Constraint_Error,
           Global => in all;

{AI12-0112-1}    procedure  Iterate                                                   60/5
     (Container : in Set;
      Process   : not null access procedure (Position : in Cursor))
      with Allows_Exit;

{AI12-0112-1}    procedure Reverse_Iterate                                            61/5
     (Container : in Set;
      Process   : not null access procedure (Position : in Cursor))
      with Allows_Exit;

{AI05-0212-1} {AI12-0112-1} {AI12-0266-1}    function Iterate (Container : in Set)   61.1/5
      return Set_Iterator_Interfaces.Parallel_Reversible_Iterator'Class
      with Post => Tampering_With_Cursors_Prohibited (Container);

{AI05-0262-1} {AI12-0112-1}    function Iterate (Container : in Set; Start : in     61.2/5
Cursor)
      return Set_Iterator_Interfaces.Reversible_Iterator'Class
      with Pre  => (Start /= No_Element
                        or else raise Constraint_Error) and then
                    (Has_Element (Container, Start)
                        or else raise Program_Error),
           Post => Tampering_With_Cursors_Prohibited (Container);

{AI12-0112-1}    generic                                                              62/5
      type Key_Type (<>) is private;
      with function Key (Element : Element_Type) return Key_Type;
      with function "<" (Left, Right : Key_Type)
         return Boolean is <>;
   package Generic_Keys
   with Nonblocking, Global => null is

       function Equivalent_Keys (Left, Right : Key_Type)                              63/2
          return Boolean;
```

```
64/5    {AI12-0112-1}       function Key (Position : Cursor) return Key_Type
                  with Pre  => Position /= No_Element or else raise Constraint_Error,
                       Global => in all;

64.1/5  {AI12-0112-1}       function Key (Container : Set;
                            Position : Cursor) return Key_Type
                  with Pre  => (Position /= No_Element
                                   or else raise Constraint_Error) and then
                               (Has_Element (Container, Position)
                                   or else raise Program_Error);

65/2          function Element (Container : Set;
                                Key       : Key_Type)
                   return Element_Type;

66/5    {AI12-0112-1}       procedure Replace (Container : in out Set;
                                Key       : in     Key_Type;
                                New_Item  : in     Element_Type)
                  with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                                   or else raise Program_Error,
                       Post => Length (Container) = Length (Container)'Old;

67/5    {AI12-0112-1}       procedure Exclude (Container : in out Set;
                                Key       : in     Key_Type)
                  with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                                   or else raise Program_Error,
                       Post => (declare
                                   Original_Length : constant Count_Type :=
                                      Length (Container)'Old;
                                begin
                                   Length (Container) in
                                      Original_Length - 1 | Original_Length);

68/5    {AI12-0112-1}       procedure Delete (Container : in out Set;
                                Key       : in     Key_Type)
                  with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                                   or else raise Program_Error,
                       Post => Length (Container) = Length (Container)'Old - 1;

69/5    {AI12-0112-1}       function Find (Container : Set;
                             Key       : Key_Type) return Cursor
                  with Post => (if Find'Result /= No_Element
                                then Has_Element (Container, Find'Result));

70/5    {AI12-0112-1}       function Floor (Container : Set;
                              Key       : Key_Type) return Cursor
                  with Post => (if Floor'Result /= No_Element
                                then Has_Element (Container, Floor'Result));

71/5    {AI12-0112-1}       function Ceiling (Container : Set;
                                Key       : Key_Type) return Cursor
                  with Post => (if Ceiling'Result /= No_Element
                                then Has_Element (Container, Ceiling'Result));

72/2          function Contains (Container : Set;
                                 Key       : Key_Type) return Boolean;

73/5    {AI12-0112-1}       procedure Update_Element_Preserving_Key
                 (Container : in out Set;
                  Position  : in     Cursor;
                  Process   : not null access procedure
                                (Element : in out Element_Type))
                  with Pre  => (Position /= No_Element
                                   or else raise Constraint_Error) and then
                               (Has_Element (Container, Position)
                                   or else raise Program_Error);

73.1/5  {AI05-0212-1} {AI12-0112-1}       type Reference_Type
                    (Element : not null access Element_Type) is private
                  with Implicit_Dereference => Element,
                       Nonblocking, Global => in out synchronized,
                       Default_Initial_Condition => (raise Program_Error);
```

```
{AI05-0212-1} {AI12-0112-1}       function Reference_Preserving_Key (Container :      73.2/5
aliased in out Set;
                                            Position  : in Cursor)
         return Reference_Type
         with Pre  => (Position /= No_Element
                         or else raise Constraint_Error) and then
                       (Has_Element (Container, Position)
                         or else raise Program_Error),
              Post => Tampering_With_Cursors_Prohibited (Container);;

{AI05-0212-1} {AI12-0112-1}       function Constant_Reference (Container : aliased   73.3/5
in Set;
                                     Key       : in Key_Type)
         return Constant_Reference_Type
         with Pre  => Find (Container, Key) /= No_Element
                         or else raise Constraint_Error,
              Post => Tampering_With_Cursors_Prohibited (Container);;

{AI05-0212-1} {AI12-0112-1}       function Reference_Preserving_Key (Container :      73.4/5
aliased in out Set;
                                           Key       : in Key_Type)
         return Reference_Type
         with Pre  => Find (Container, Key) /= No_Element
                         or else raise Constraint_Error,
              Post => Tampering_With_Cursors_Prohibited (Container);;

   end Generic_Keys;                                                               74/2

{AI12-0111-1}    package Stable is                                                  74.1/5

{AI12-0111-1} {AI12-0339-1} {AI12-0399-1} {AI12-0407-1} {AI12-0445-1}       type  74.2/5
Set (Base : not null access Ordered_Sets.Set) is
         tagged limited private
         with Constant_Indexing => Constant_Reference,
              Default_Iterator  => Iterate,
              Iterator_Element  => Element_Type,
              Stable_Properties => (Length),
              Global            => null,
              Default_Initial_Condition => Length (Set) = 0,
              Preelaborable_Initialization;

{AI12-0111-1} {AI12-0399-1}       type Cursor is private                          74.3/5
         with Preelaborable_Initialization;

{AI12-0111-1}       Empty_Set : constant Set;                                     74.4/5

{AI12-0111-1}       No_Element : constant Cursor;                                 74.5/5

{AI12-0111-1}       function Has_Element (Position : Cursor) return Boolean       74.6/5
         with Nonblocking, Global => in all, Use_Formal => null;

{AI12-0111-1}       package Set_Iterator_Interfaces is new                        74.7/5
         Ada.Iterator_Interfaces (Cursor, Has_Element);

{AI12-0111-1} {AI12-0445-1}       procedure Assign (Target : in out              74.8/5
Ordered_Sets.Set;
                        Source : in Set)
         with Post => Length (Source) = Length (Target);

{AI12-0111-1} {AI12-0445-1}       function Copy (Source : Ordered_Sets.Set) return  74.9/5
Set
         with Post => Length (Copy'Result) = Length (Source);

{AI12-0111-1}       type Constant_Reference_Type                                  74.10/5
            (Element : not null access constant Element_Type) is private
         with Implicit_Dereference => Element,
              Nonblocking, Global => null, Use_Formal => null,
              Default_Initial_Condition => (raise Program_Error);

{AI12-0111-1}       -- Additional subprograms as described in the text            74.11/5
      -- are declared here.

{AI12-0111-1}    private                                                          74.12/5

{AI12-0111-1}       ... -- not specified by the language                          74.13/5
```

74.14/5
```
{AI12-0111-1}    end Stable;
```

75/2
```
private
```

76/2
```
   ... -- not specified by the language
```

77/2
```
end Ada.Containers.Ordered_Sets;
```

78/2 {*AI95-00302-03*} Two elements *E1* and *E2* are *equivalent* if both *E1* < *E2* and *E2* < *E1* return False, using the generic formal "<" operator for elements. Function Equivalent_Elements returns True if Left and Right are equivalent, and False otherwise.

79/3 {*AI95-00302-03*} {*AI05-0044-1*} The actual function for the generic formal function "<" on Element_Type values is expected to return the same value each time it is called with a particular pair of key values. It should define a strict weak ordering relationship (see A.18). If the actual for "<" behaves in some other manner, the behavior of this package is unspecified. Which subprograms of this package call "<" and how many times they call it, is unspecified.

79.1/3 {*AI05-0044-1*} If the actual function for the generic formal function "=" returns True for any pair of nonequivalent elements, then the behavior of the container function "=" is unspecified.

80/2 {*AI95-00302-03*} If the value of an element stored in a set is changed other than by an operation in this package such that at least one of "<" or "=" give different results, the behavior of this package is unspecified.

80.a/2 **Discussion:** See A.18.6, “The Generic Package Containers.Ordered_Maps” for a suggested implementation, and for justification of the restrictions regarding "<" and "=". Note that sets only need to store elements, not key/element pairs.

81/3 {*AI95-00302-03*} {*AI05-0262-1*} The *first element* of a nonempty set is the one which is less than all the other elements in the set. The *last element* of a nonempty set is the one which is greater than all the other elements in the set. The *successor* of an element is the smallest element that is larger than the given element. The *predecessor* of an element is the largest element that is smaller than the given element. All comparisons are done using the generic formal "<" operator for elements.

81.1/5
```
{AI12-0112-1} function Copy (Source : Set) return Set
   with Post => Length (Copy'Result) = Length (Source) and then
                not Tampering_With_Cursors_Prohibited (Copy'Result);
```

81.2/3 {*AI05-0001-1*} Returns a set whose elements are initialized from the corresponding elements of Source.

82/5
```
{AI12-0112-1} procedure Delete_First (Container : in out Set)
   with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                   or else raise Program_Error,
        Post => (declare
                   Original_Length : constant Count_Type :=
                      Length (Container)'Old;
                 begin
                   (if Original_Length = 0 then Length (Container) = 0
                    else Length (Container) = Original_Length - 1));
```

83/3 {*AI95-00302-03*} {*AI05-0264-1*} If Container is empty, Delete_First has no effect. Otherwise, the element designated by First (Container) is removed from Container. Delete_First tampers with the cursors of Container.

```
{AI12-0112-1} procedure Delete_Last (Container : in out Set)                    84/5
   with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                  or else raise Program_Error,
        Post => (declare
                   Original_Length : constant Count_Type :=
                      Length (Container)'Old;
                 begin
                   (if Original_Length = 0 then Length (Container) = 0
                    else Length (Container) = Original_Length - 1));
```

{*AI95-00302-03*} {*AI05-0264-1*} If Container is empty, Delete_Last has no effect. Otherwise, the element designated by Last (Container) is removed from Container. Delete_Last tampers with the cursors of Container. 85/3

```
{AI12-0112-1} function First_Element (Container : Set) return Element_Type      86/5
   with Pre => (not Is_Empty (Container)
                  or else raise Constraint_Error);
```

{*AI95-00302-03*} Equivalent to Element (First (Container)). 87/2

```
{AI12-0112-1} function Last (Container : Set) return Cursor                     88/5
   with Nonblocking, Global => null, Use_Formal => null,
        Post => (if not Is_Empty (Container) then
                    Has_Element (Container, Last'Result)
                 else Last'Result = No_Element);
```

{*AI95-00302-03*} Returns a cursor that designates the last element in Container. If Container is empty, returns No_Element. 89/2

```
{AI12-0112-1} function Last_Element (Container : Set) return Element_Type       90/5
   with Pre => (not Is_Empty (Container)
                  or else raise Constraint_Error);
```

{*AI95-00302-03*} Equivalent to Element (Last (Container)). 91/2

```
{AI12-0112-1} function Previous (Position : Cursor) return Cursor               92/5
   with Nonblocking, Global => in all, Use_Formal => null,
        Post => (if Position = No_Element then
                   Previous'Result = No_Element);
```

{*AI95-00302-03*} {*AI05-0262-1*} If Position equals No_Element, then Previous returns No_Element. Otherwise, Previous returns a cursor designating the predecessor element of the one designated by Position. If Position designates the first element, then Previous returns No_Element. 93/3

```
function Previous (Container : Set;                                             93.1/5
                   Position : Cursor) return Cursor
   with Nonblocking, Global => null, Use_Formal => null,
        Pre  => Position = No_Element or else
                Has_Element (Container, Position)
                  or else raise Program_Error,
        Post => (if Position = No_Element then
                   Previous'Result = No_Element
                 elsif Previous'Result = No_Element then
                   Position = First (Container)
                 else Has_Element (Container, Previous'Result));
```

{*AI12-0112-1*} Returns a cursor designating the predecessor of the node designated by Position in Container, if any. 93.2/5

```
{AI12-0112-1} procedure Previous (Position : in out Cursor)                     94/5
   with Nonblocking, Global => in all, Use_Formal => null;
```

{*AI95-00302-03*} Equivalent to Position := Previous (Position). 95/2

95.1/5
```
procedure Previous (Container : in     Set;
                    Position  : in out Cursor)
   with Nonblocking, Global => null, Use_Formal => null,
        Pre  => Position = No_Element or else
                Has_Element (Container, Position)
                   or else raise Program_Error,
        Post => (if Position /= No_Element
                 then Has_Element (Container, Position));
```

95.2/5 {*AI12-0112-1*} Equivalent to Position := Previous (Container, Position).

96/5
```
{AI12-0112-1} function Floor (Container : Set;
                              Item      : Element_Type) return Cursor
   with Post => (if Floor'Result /= No_Element
                 then Has_Element (Container, Floor'Result));
```

97/3 {*AI95-00302-03*} {*AI05-0264-1*} Floor searches for the last element which is not greater than Item. If such an element is found, a cursor that designates it is returned. Otherwise, No_Element is returned.

98/5
```
{AI12-0112-1} function Ceiling (Container : Set;
                                Item      : Element_Type) return Cursor
   with Post => (if Ceiling'Result /= No_Element
                 then Has_Element (Container, Ceiling'Result));
```

99/3 {*AI95-00302-03*} {*AI05-0264-1*} Ceiling searches for the first element which is not less than Item. If such an element is found, a cursor that designates it is returned. Otherwise, No_Element is returned.

100/5
```
{AI12-0112-1} function "<" (Left, Right : Cursor) return Boolean
   with Pre    => (Left /= No_Element and then Right /= No_Element)
                      or else raise Constraint_Error,
        Global => in all;
```

101/2 {*AI95-00302-03*} Equivalent to Element (Left) < Element (Right).

102/5
```
{AI12-0112-1} function ">" (Left, Right : Cursor) return Boolean
   with Pre    => (Left /= No_Element and then Right /= No_Element)
                      or else raise Constraint_Error,
        Global => in all;
```

103/2 {*AI95-00302-03*} Equivalent to Element (Right) < Element (Left).

104/5
```
{AI12-0112-1} function "<" (Left : Cursor; Right : Element_Type) return Boolean
   with Pre    => Left /= No_Element or else raise Constraint_Error,
        Global => in all;
```

105/2 {*AI95-00302-03*} Equivalent to Element (Left) < Right.

106/5
```
{AI12-0112-1} function ">" (Left : Cursor; Right : Element_Type) return Boolean
   with Pre    => Left /= No_Element or else raise Constraint_Error,
          Global => in all;
```

107/2 {*AI95-00302-03*} Equivalent to Right < Element (Left).

108/5
```
{AI12-0112-1} function "<" (Left : Element_Type; Right : Cursor) return Boolean
   with Pre    => Right /= No_Element or else raise Constraint_Error,
          Global => in all;
```

109/2 {*AI95-00302-03*} Equivalent to Left < Element (Right).

110/5
```
{AI12-0112-1} function ">" (Left : Element_Type; Right : Cursor) return Boolean
   with Pre    => Right /= No_Element or else raise Constraint_Error,
          Global => in all;
```

111/2 {*AI95-00302-03*} Equivalent to Element (Right) < Left.

```
{AI12-0112-1} procedure Reverse_Iterate                                    112/5
  (Container : in Set;
   Process   : not null access procedure (Position : in Cursor))
   with Allows_Exit;
```

{*AI95-00302-03*} {*AI05-0212-1*} Iterates over the elements in Container as per procedure Iterate, with the difference that the elements are traversed in predecessor order, starting with the last element. 113/3

```
{AI12-0112-1} {AI12-0266-1} function Iterate (Container : in Set)          113.1/5
   return Set_Iterator_Interfaces.Parallel_Reversible_Iterator'Class
   with Post => Tampering_With_Cursors_Prohibited (Container);
```

{*AI05-0212-1*} {*AI05-0265-1*} {*AI05-0269-1*} {*AI12-0266-1*} Iterate returns an iterator object (see 5.5.1) that will generate a value for a loop parameter (see 5.5.2) designating each element in Container, starting with the first element and moving the cursor according to the successor relation when used as a forward iterator, and starting with the last element and moving the cursor according to the predecessor relation when used as a reverse iterator, and processing all nodes concurrently when used as a parallel iterator. Tampering with the cursors of Container is prohibited while the iterator object exists (in particular, in the sequence_of_statements of the loop_statement whose iterator_specification denotes this object). The iterator object needs finalization. 113.2/5

```
function Iterate (Container : in Set; Start : in Cursor)                  113.3/5
   return Set_Iterator_Interfaces.Reversible_Iterator'Class
   with Pre  => (Start /= No_Element
                    or else raise Constraint_Error) and then
                (Has_Element (Container, Start)
                    or else raise Program_Error),
        Post => Tampering_With_Cursors_Prohibited (Container);
```

{*AI05-0262-1*} {*AI05-0265-1*} {*AI05-0269-1*} {*AI12-0112-1*} Iterate returns a reversible iterator object (see 5.5.1) that will generate a value for a loop parameter (see 5.5.2) designating each element in Container, starting with the element designated by Start and moving the cursor according to the successor relation when used as a forward iterator, or moving the cursor according to the predecessor relation when used as a reverse iterator. Tampering with the cursors of Container is prohibited while the iterator object exists (in particular, in the sequence_of_statements of the loop_statement whose iterator_specification denotes this object). The iterator object needs finalization. 113.4/5

Discussion: Exits are allowed from the loops created using the iterator objects. In particular, to stop the iteration at a particular cursor, just add 113.a/3

```
exit when Cur = Stop;                                                      113.b/3
```

in the body of the loop (assuming that `Cur` is the loop parameter and `Stop` is the cursor that you want to stop at). 113.c/3

{*AI95-00302-03*} For any two elements *E1* and *E2*, the boolean values (*E1* < *E2*) and (Key(*E1*) < Key(*E2*)) are expected to be equal. If the actuals for Key or Generic_Keys."<" behave in some other manner, the behavior of this package is unspecified. Which subprograms of this package call Key and Generic_Keys."<", and how many times the functions are called, is unspecified. 114/2

{*AI95-00302-03*} In addition to the semantics described in A.18.7, the subprograms in package Generic_Keys named Floor and Ceiling, are equivalent to the corresponding subprograms in the parent package, with the difference that the Key subprogram parameter is compared to elements in the container using the Key and "<" generic formal functions. The function named Equivalent_Keys in package Generic_Keys returns True if both Left < Right and Right < Left return False using the generic formal "<" operator, and returns True otherwise. 115/2

Implementation Advice

116/2 {*AI95-00302-03*} If *N* is the length of a set, then the worst-case time complexity of the Insert, Include, Replace, Delete, Exclude, and Find operations that take an element parameter should be *O*((log *N*)**2) or better. The worst-case time complexity of the subprograms that take a cursor parameter should be *O*(1).

116.a/2 **Implementation Advice:** The worst-case time complexity of the Insert, Include, Replace, Delete, Exclude, and Find operations of Containers.Ordered_Sets that take an element parameter should be *O*((log *N*)**2). The worst-case time complexity of the subprograms of Containers.Ordered_Sets that take a cursor parameter should be *O*(1).

116.b/2 **Implementation Note:** {*AI95-00302-03*} See A.18.6, "The Generic Package Containers.Ordered_Maps" for implementation notes regarding some of the operations of this package.

Extensions to Ada 95

116.c/2 {*AI95-00302-03*} The generic package Containers.Ordered_Sets is new.

Incompatibilities With Ada 2005

116.d/3 {*AI05-0001-1*} Subprograms Assign and Copy are added to Containers.Ordered_Sets. If an instance of Containers.Ordered_Sets is referenced in a use_clause, and an entity *E* with the same defining_identifier as a new entity in Containers.Ordered_Sets is defined in a package that is also referenced in a use_clause, the entity *E* may no longer be use-visible, resulting in errors. This should be rare and is easily fixed if it does occur.

Extensions to Ada 2005

116.e/3 {*AI05-0212-1*} Added iterator and indexing support to make ordered set containers more convenient to use.

Wording Changes from Ada 2005

116.f/3 {*AI05-0044-1*} **Correction:** Added wording to require the formal function be such that equal elements are also equivalent.

116.g/3 {*AI05-0044-1*} **Correction:** Redefined "<" actuals to require a strict weak ordering; the old definition allowed indeterminant comparisons that would not have worked in a container.

116.h/3 {*AI05-0084-1*} **Correction:** Added a pragma Remote_Types so that containers can be used in distributed programs.

Incompatibilities With Ada 2012

116.i/5 {*AI12-0111-1*} {*AI12-0112-1*} {*AI12-0339-1*} A number of new subprograms, types, and even a nested package were added to Containers.Ordered_Sets to better support contracts and stable views. Therefore, a use clause conflict is possible; see the introduction of Annex A for more on this topic.

Extensions to Ada 2012

116.j/5 {*AI12-0212-1*} Sets now support positional container aggregates, so aggregate syntax can be used to create Sets.

116.k/5 {*AI12-0266-1*} The iterator for the entire container now can return a parallel iterator which can be used to process the container in parallel.

Wording Changes from Ada 2012

116.l/5 {*AI12-0112-1*} Added contracts to this container. This includes describing some of the semantics with pre- and postconditions, rather than English text. Note that the preconditions can be Suppressed (see 11.5).

A.18.10 The Generic Package Containers.Multiway_Trees

1/3 {*AI05-0136-1*} The language-defined generic package Containers.Multiway_Trees provides private types Tree and Cursor, and a set of operations for each type. A multiway tree container is well-suited to represent nested structures.

1.a/3 **Discussion:** {*AI05-0136-1*} This tree just provides a basic structure, and make no promises about balancing or other automatic organization. In this sense, it is different than the indexed (Map, Set) forms. Rather, it provides a building block on which to construct more complex and more specialized tree containers.

2/4 {*AI05-0136-1*} {*AI12-0078-1*} {*AI12-0159-1*} A multiway tree container object manages a tree of *nodes*, consisting of a *root node* and a set of *internal nodes*; each internal node contains an element and pointers

to the parent, first child, last child, next (successor) sibling, and previous (predecessor) sibling internal nodes. A cursor designates a particular node within a tree (and by extension the element contained in that node, if any). A cursor keeps designating the same node (and element) as long as the node is part of the container, even if the node is moved within the container.

{*AI05-0136-1*} {*AI05-0269-1*} {*AI12-0078-1*} A *subtree* is a particular node (which *roots the subtree*) and all of its child nodes (including all of the children of the child nodes, recursively). The root node is always present and has neither an associated element value nor any parent node; it has pointers to its first child and its last child, if any. The root node provides a place to add nodes to an otherwise empty tree and represents the base of the tree. 3/4

{*AI05-0136-1*} {*AI05-0269-1*} A node that has no children is called a *leaf node*. The *ancestors* of a node are the node itself, its parent node, the parent of the parent node, and so on until a node with no parent is reached. Similarly, the *descendants* of a node are the node itself, its child nodes, the children of each child node, and so on. 4/3

{*AI05-0136-1*} {*AI05-0262-1*} {*AI05-0269-1*} The nodes of a subtree can be visited in several different orders. For a *depth-first order*, after visiting a node, the nodes of its child list are each visited in depth-first order, with each child node visited in natural order (first child to last child). 5/3

Ramification: For the depth-first order, when each child node is visited, the child list of the child node is visited before the next sibling of the child node is visited. 5.a/3

Static Semantics

{*AI05-0136-1*} The generic library package Containers.Multiway_Trees has the following declaration: 6/3

```
{AI05-0136-1} {AI05-0212-1} {AI12-0112-1} with Ada.Iterator_Interfaces;                    7/5
generic
   type Element_Type is private;
   with function "=" (Left, Right : Element_Type) return Boolean is <>;
package Ada.Containers.Multiway_Trees
   with Preelaborate, Remote_Types,
        Nonblocking, Global => in out synchronized is
```

Discussion: {*AI12-0112-1*} For discussion on the reasons and meaning of the specifications of the Global and Nonblocking aspects in this generic package, see the notes on the equivalent operations in the specification of the Containers.Vectors package (see A.18.2). 7.a/5

```
{AI05-0136-1} {AI05-0212-1} {AI12-0111-1} {AI12-0112-1} {AI12-0399-1}    type Tree    8/5
is tagged private
      with Constant_Indexing => Constant_Reference,
           Variable_Indexing => Reference,
           Default_Iterator  => Iterate,
           Iterator_Element  => Element_Type,
           Iterator_View     => Stable.Tree,
           Stable_Properties => (Node_Count,
                                 Tampering_With_Cursors_Prohibited,
                                 Tampering_With_Elements_Prohibited),
           Default_Initial_Condition =>
              Node_Count (Tree) = 1 and then
              (not Tampering_With_Cursors_Prohibited (Tree)) and then
              (not Tampering_With_Elements_Prohibited (Tree)),
           Preelaborable_Initialization;

{AI12-0399-1}    type Cursor is private                                                    9/5
      with Preelaborable_Initialization;

   Empty_Tree : constant Tree;                                                           10/3

   No_Element : constant Cursor;                                                         11/3

{AI12-0112-1}    function Equal_Element (Left, Right : Element_Type)                   11.1/5
      return Boolean renames "=";
```

12/5 {*AI05-0212-1*} {*AI12-0112-1*} **function** Has_Element (Position : Cursor) **return** Boolean
with Nonblocking, Global => **in all**, Use_Formal => **null**;

12.1/5 {*AI12-0112-1*} **function** Has_Element (Container : Tree; Position : Cursor)
return Boolean
with Nonblocking, Global => **null**, Use_Formal => **null**;

13/3 {*AI05-0212-1*} **package** Tree_Iterator_Interfaces **is new**
Ada.Iterator_Interfaces (Cursor, Has_Element);

14/3 **function** Equal_Subtree (Left_Position : Cursor;
Right_Position: Cursor) **return** Boolean;

15/3 **function** "=" (Left, Right : Tree) **return** Boolean;

15.1/5 {*AI12-0112-1*} **function** Tampering_With_Cursors_Prohibited
(Container : Tree) **return** Boolean
with Nonblocking, Global => **null**, Use_Formal => **null**;

15.2/5 {*AI12-0112-1*} **function** Tampering_With_Elements_Prohibited
(Container : Tree) **return** Boolean
with Nonblocking, Global => **null**, Use_Formal => **null**;

15.3/5 {*AI12-0339-1*} **function** Empty **return** Tree
is (Empty_Tree)
with Post =>
not Tampering_With_Elements_Prohibited (Empty'Result) **and then**
not Tampering_With_Cursors_Prohibited (Empty'Result) **and then**
Node_Count (Empty'Result) = 1;

16/5 {*AI12-0112-1*} **function** Is_Empty (Container : Tree) **return** Boolean
with Nonblocking, Global => **null**, Use_Formal => **null**,
Post => Is_Empty'Result = (Node_Count (Container) = 1);

17/5 {*AI12-0112-1*} **function** Node_Count (Container : Tree) **return** Count_Type
with Nonblocking, Global => **null**, Use_Formal => **null**;

18/5 {*AI12-0112-1*} **function** Subtree_Node_Count (Position : Cursor) **return** Count_Type
with Nonblocking, Global => **in all**, Use_Formal => **null**;

18.1/5 {*AI12-0112-1*} **function** Subtree_Node_Count (Container : Tree; Position : Cursor)
return Count_Type
with Pre => Meaningful_For (Container, Position)
or else raise Program_Error,
Nonblocking, Global => **null**, Use_Formal => **null**;

19/5 {*AI12-0112-1*} **function** Depth (Position : Cursor) **return** Count_Type
with Nonblocking, Global => **in all**, Use_Formal => **null**;

19.1/5 {*AI12-0112-1*} **function** Depth (Container : Tree; Position : Cursor)
return Count_Type
with Pre => Meaningful_For (Container, Position)
or else raise Program_Error,
Nonblocking, Global => **null**, Use_Formal => **null**;

20/5 {*AI12-0112-1*} **function** Is_Root (Position : Cursor) **return** Boolean
with Nonblocking, Global => **in all**, Use_Formal => **null**;

20.1/5 {*AI12-0112-1*} **function** Is_Root (Container : Tree; Position : Cursor)
return Boolean
with Nonblocking, Global => **null**, Use_Formal => **null**;

21/5 {*AI12-0112-1*} **function** Is_Leaf (Position : Cursor) **return** Boolean
with Nonblocking, Global => **in all**, Use_Formal => **null**;

21.1/5 {*AI12-0112-1*} **function** Is_Leaf (Container : Tree; Position : Cursor)
return Boolean
with Pre => Meaningful_For (Container, Position)
or else raise Program_Error,
Nonblocking, Global => **null**, Use_Formal => **null**;

21.2/5
```
{AI12-0112-1}    function Is_Ancestor_Of (Container : Tree;
                                          Parent   : Cursor;
                                          Position : Cursor) return Boolean
      with Pre => (Meaningful_For (Container, Position)
                      or else raise Program_Error) and then
                   (Meaningful_For (Container, Parent)
                      or else raise Program_Error),
           Nonblocking, Global => null, Use_Formal => null;
```

22/5
```
{AI12-0112-1}    function Root (Container : Tree) return Cursor
      with Nonblocking, Global => null, Use_Formal => null,
           Post => Root'Result /= No_Element and then
                   not Has_Element (Container, Root'Result);
```

22.1/5
```
{AI12-0112-1}    function Meaningful_For (Container : Tree; Position : Cursor)
      return Boolean is
      (Position = No_Element or else
       Is_Root (Container, Position) or else
       Has_Element (Container, Position))
      with Nonblocking, Global => null, Use_Formal => null;
```

Reason: {*AI12-0112-1*} When this function is true, the Position can be meaningfully used with operations for Container. We define this because many operations allow the root (which does not have an element, so Has_Element returns False), so many preconditions get unwieldy. We allow No_Element as it is allowed by many queries, and for existing routines, it raises a different exception (Constraint_Error rather than Program_Error) than a cursor for the wrong container does. 22.a/5

23/5
```
{AI12-0112-1}    procedure Clear (Container : in out Tree)
      with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                      or else raise Program_Error,
           Post => Node_Count (Container) = 1;
```

24/5
```
{AI12-0112-1}    function Element (Position : Cursor) return Element_Type
      with Pre => (Position /= No_Element or else
                      raise Constraint_Error) and then
                  (Has_Element (Position) or else raise Program_Error),
           Nonblocking, Global => in all, Use_Formal => Element_Type;
```

24.1/5
```
{AI12-0112-1}    function Element (Container : Tree;
                                   Position  : Cursor) return Element_Type
      with Pre => (Position /= No_Element or else
                      raise Constraint_Error) and then
                  (Has_Element (Container, Position)
                     or else raise Program_Error),
           Nonblocking, Global => null, Use_Formal => Element_Type;
```

25/5
```
{AI12-0112-1}    procedure Replace_Element (Container : in out Tree;
                                            Position  : in     Cursor;
                                            New_item  : in     Element_Type)
      with Pre  => (not Tampering_With_Elements_Prohibited (Container)
                     or else raise Program_Error) and then
                   (Position /= No_Element
                     or else raise Constraint_Error) and then
                   (Has_Element (Container, Position)
                     or else raise Program_Error);
```

26/5
```
{AI12-0112-1}    procedure Query_Element
     (Position : in Cursor;
      Process  : not null access procedure (Element : in Element_Type))
      with Pre  => (Position /= No_Element
                      or else raise Constraint_Error) and then
                   (Has_Element (Position) or else raise Program_Error),
           Global => in all;
```

26.1/5
```
{AI12-0112-1}    procedure Query_Element
     (Container : in Tree;
      Position  : in Cursor;
      Process   : not null access procedure (Element : in Element_Type))
      with Pre  => (Position /= No_Element
                      or else raise Constraint_Error) and then
                   (Has_Element (Container, Position)
                     or else raise Program_Error);
```

27/5 {*AI12-0112-1*} **procedure** Update_Element
(Container : **in out** Tree;
Position : **in** Cursor;
Process : **not null access procedure**
(Element : **in out** Element_Type))
with Pre => (Position /= No_Element
or else raise Constraint_Error) **and then**
(Has_Element (Container, Position)
or else raise Program_Error);

28/5 {*AI05-0212-1*} {*AI12-0112-1*} **type** Constant_Reference_Type
(Element : **not null access constant** Element_Type) **is private**
with Implicit_Dereference => Element,
Nonblocking, Global => **in out synchronized**,
Default_Initial_Condition => (**raise** Program_Error);

29/5 {*AI05-0212-1*} {*AI12-0112-1*} **type** Reference_Type (Element : **not null access** Element_Type) **is private**
with Implicit_Dereference => Element,
Nonblocking, Global => **in out synchronized**,
Default_Initial_Condition => (**raise** Program_Error);

30/5 {*AI05-0212-1*} {*AI12-0112-1*} **function** Constant_Reference (Container : **aliased in** Tree;
Position : **in** Cursor)
return Constant_Reference_Type
with Pre => (Position /= No_Element
or else raise Constraint_Error) **and then**
(Has_Element (Container, Position)
or else raise Program_Error),
Post => Tampering_With_Cursors_Prohibited (Container),
Nonblocking, Global => **null**, Use_Formal => **null**;

31/5 {*AI05-0212-1*} {*AI12-0112-1*} **function** Reference (Container : **aliased in out** Tree;
Position : **in** Cursor)
return Reference_Type
with Pre => (Position /= No_Element
or else raise Constraint_Error) **and then**
(Has_Element (Container, Position)
or else raise Program_Error),
Post => Tampering_With_Cursors_Prohibited (Container),
Nonblocking, Global => **null**, Use_Formal => **null**;

32/5 {*AI12-0112-1*} **procedure** Assign (Target : **in out** Tree; Source : **in** Tree)
with Pre => **not** Tampering_With_Cursors_Prohibited (Target)
or else raise Program_Error,
Post => Node_Count (Source) = Node_Count (Target);

33/5 {*AI12-0112-1*} **function** Copy (Source : Tree) **return** Tree
with Post =>
Node_Count (Copy'Result) = Node_Count (Source) **and then**
not Tampering_With_Elements_Prohibited (Copy'Result) **and then**
not Tampering_With_Cursors_Prohibited (Copy'Result);

34/5 {*AI12-0112-1*} **procedure** Move (Target : **in out** Tree;
Source : **in out** Tree)
with Pre => (**not** Tampering_With_Cursors_Prohibited (Target)
or else raise Program_Error) **and then**
(**not** Tampering_With_Cursors_Prohibited (Source)
or else raise Program_Error),
Post => (**if not** Target'Has_Same_Storage (Source) **then**
Node_Count (Target) = Node_Count (Source'Old) **and then**
Node_Count (Source) = 1);

```
{AI12-0112-1}    procedure Delete_Leaf (Container : in out Tree;                    35/5
                          Position  : in out Cursor)
      with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                       or else raise Program_Error) and then
                    (Position /= No_Element
                       or else raise Constraint_Error) and then
                    (Has_Element (Container, Position)
                       or else raise Program_Error) and then
                    (Is_Leaf (Container, Position)
                       or else raise Constraint_Error),
           Post =>
              Node_Count (Container)'Old = Node_Count (Container)+1 and then
              Position = No_Element;

{AI12-0112-1}    procedure Delete_Subtree (Container : in out Tree;                 36/5
                             Position  : in out Cursor)
      with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                       or else raise Program_Error) and then
                    (Position /= No_Element
                       or else raise Constraint_Error) and then
                    (Has_Element (Container, Position)
                       or else raise Program_Error),
           Post => Node_Count (Container)'Old = Node_Count (Container) +
                       Subtree_Node_Count (Container, Position)'Old and then
                    Position = No_Element;

{AI12-0112-1}    procedure Swap (Container : in out Tree;                           37/5
                    I, J      : in     Cursor)
      with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                       or else raise Program_Error) and then
                    (I /= No_Element or else Constraint_Error) and then
                    (J /= No_Element or else Constraint_Error) and then
                    (Has_Element (Container, I)
                       or else raise Program_Error) and then
                    (Has_Element (Container, J)
                       or else raise Program_Error);

{AI12-0112-1}    function Find (Container : Tree;                                   38/5
                     Item      : Element_Type)
      return Cursor
      with Post => (if Find'Result /= No_Element
                     then Has_Element (Container, Find'Result));

{AI05-0136-1} {AI05-0248-1} {AI12-0112-1}    function Find_In_Subtree (Position :  39/5
Cursor;
                             Item     : Element_Type)
      return Cursor
      with Pre  => Position /= No_Element or else raise Constraint_Error,
           Post => (if Find_In_Subtree'Result = No_Element
                     then Has_Element (Find_In_Subtree'Result)),
           Global => in all;

{AI12-0112-1}    function Find_In_Subtree (Container : Tree;                        39.1/5
                             Position  : Cursor;
                             Item      : Element_Type)
      return Cursor
      with Pre  => (Position /= No_Element
                       or else raise Constraint_Error) and then
                    (Meaningful_For (Container, Position)
                       or else raise Program_Error),
           Post => (if Find_In_Subtree'Result /= No_Element
                     then Has_Element (Container, Find_In_Subtree'Result));

{AI05-0136-1} {AI05-0248-1} {AI12-0112-1}    function Ancestor_Find (Position :  40/5
Cursor;
                           Item     : Element_Type)
      return Cursor
      with Pre  => Position /= No_Element or else raise Constraint_Error,
           Post => (if Ancestor_Find'Result = No_Element
                     then Has_Element (Ancestor_Find'Result)),
           Global => in all;
```

40.1/5
```
{AI12-0112-1}    function Ancestor_Find (Container : Tree;
                                         Position  : Cursor;
                                         Item      : Element_Type)
      return Cursor
      with Pre  => (Position /= No_Element
                       or else raise Constraint_Error) and then
                   (Meaningful_For (Container, Position)
                       or else raise Program_Error),
           Post => (if Ancestor_Find'Result = No_Element
                    then Has_Element (Container, Ancestor_Find'Result));
```

41/3
```
   function Contains (Container : Tree;
                      Item      : Element_Type) return Boolean;
```

42/5
```
{AI12-0112-1}    procedure Iterate
     (Container : in Tree;
      Process   : not null access procedure (Position : in Cursor))
      with Allows_Exit;
```

43/5
```
{AI12-0112-1}    procedure Iterate_Subtree
     (Position  : in Cursor;
      Process   : not null access procedure (Position : in Cursor))
      with Allows_Exit,
           Pre  => Position /= No_Element or else raise Constraint_Error,
           Global => in all;
```

43.1/5
```
{AI12-0112-1}    procedure Iterate_Subtree
     (Container : in Tree;
      Position  : in Cursor;
      Process   : not null access procedure (Position : in Cursor))
      with Allows_Exit,
           Pre  => (Position /= No_Element
                       or else raise Constraint_Error) and then
                   (Meaningful_For (Container, Position)
                       or else raise Program_Error);
```

44/5
```
{AI05-0212-1} {AI12-0112-1} {AI12-0266-1}    function Iterate (Container : in Tree)
      return Tree_Iterator_Interfaces.Parallel_Iterator'Class
      with Post => Tampering_With_Cursors_Prohibited (Container);
```

45/5
```
{AI05-0212-1} {AI12-0112-1} {AI12-0266-1}    function Iterate_Subtree (Position :
in Cursor)
      return Tree_Iterator_Interfaces.Parallel_Iterator'Class
      with Pre    => Position /= No_Element or else raise Constraint_Error,
           Global => in all;
```

45.1/5
```
{AI12-0112-1}    function Iterate_Subtree (Container : in Tree; Position : in
Cursor)
      return Tree_Iterator_Interfaces.Parallel_Iterator'Class
      with Pre  => (Position /= No_Element
                       or else raise Constraint_Error) and then
                   (Meaningful_For (Container, Position)
                       or else raise Program_Error),
           Post => Tampering_With_Cursors_Prohibited (Container);
```

46/5
```
{AI12-0112-1}    function Child_Count (Parent : Cursor) return Count_Type
      with Post => (if Parent = No_Element then Child_Count'Result = 0),
      with Nonblocking, Global => in all, Use_Formal => null;
```

46.1/5
```
{AI12-0112-1}    function Child_Count (Container : Tree; Parent : Cursor)
      return Count_Type
      with Pre  => Meaningful_For (Container, Parent)
                       or else raise Program_Error,
           Post => (if Parent = No_Element then Child_Count'Result = 0),
           Nonblocking, Global => null, Use_Formal => null;
```

47/5
```
{AI12-0112-1}    function Child_Depth (Parent, Child : Cursor) return Count_Type
      with Pre  => (Parent = No_Element and then Child = No_Element)
                       or else raise Constraint_Error,
      with Nonblocking, Global => in all, Use_Formal => null;
```

```
{AI12-0112-1}    function Child_Depth (Container : Tree; Parent, Child : Cursor)      47.1/5
      return Count_Type
      with Pre  => ((Parent = No_Element and then Child = No_Element)
                       or else raise Constraint_Error) and then
                    (Meaningful_For (Container, Parent)
                       or else raise Program_Error) and then
                    (Meaningful_For (Container, Child)
                       or else raise Program_Error),
           Nonblocking, Global => null, Use_Formal => null;

{AI12-0112-1}    procedure Insert_Child (Container : in out Tree;                    48/5
                           Parent    : in     Cursor;
                           Before    : in     Cursor;
                           New_Item  : in     Element_Type;
                           Count     : in     Count_Type := 1)
      with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                       or else raise Program_Error) and then
                    (Parent /= No_Element
                       or else raise Constraint_Error) and then
                    (Meaningful_For (Container, Parent)
                       or else raise Program_Error) and then
                    (Meaningful_For (Container, Before)
                       or else raise Program_Error) and then
                    (Before = No_Element or else
                     Container.Parent (Before) = Parent
                       or else raise Constraint_Error),
           Post => Node_Count (Container) =
                   Node_Count (Container)'Old + Count;

{AI12-0112-1}    procedure Insert_Child (Container : in out Tree;                    49/5
                           Parent    : in     Cursor;
                           Before    : in     Cursor;
                           New_Item  : in     Element_Type;
                           Position  :    out Cursor;
                           Count     : in     Count_Type := 1)
      with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                       or else raise Program_Error) and then
                    (Parent /= No_Element
                       or else raise Constraint_Error) and then
                    (Meaningful_For (Container, Parent)
                       or else raise Program_Error) and then
                    (Meaningful_For (Container, Before)
                       or else raise Program_Error) and then
                    (Before = No_Element or else
                     Container.Parent (Before) = Parent
                       or else raise Constraint_Error),
           Post => (Node_Count (Container) =
                    Node_Count (Container)'Old + Count) and then
                    Has_Element (Container, Position);

{AI12-0112-1}    procedure Insert_Child (Container : in out Tree;                    50/5
                           Parent    : in     Cursor;
                           Before    : in     Cursor;
                           Position  :    out Cursor;
                           Count     : in     Count_Type := 1)
      with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                       or else raise Program_Error) and then
                    (Parent /= No_Element
                       or else raise Constraint_Error) and then
                    (Meaningful_For (Container, Parent)
                       or else raise Program_Error) and then
                    (Meaningful_For (Container, Before)
                       or else raise Program_Error) and then
                    (Before = No_Element or else
                     Container.Parent (Before) = Parent
                       or else raise Constraint_Error),
           Post => (Node_Count (Container) =
                    Node_Count (Container)'Old + Count) and then
                    Has_Element (Container, Position);
```

51/5 {*AI12-0112-1*}
```
procedure Prepend_Child (Container : in out Tree;
                         Parent    : in     Cursor;
                         New_Item  : in     Element_Type;
                         Count     : in     Count_Type := 1)
      with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                      or else raise Program_Error) and then
                    (Parent /= No_Element
                      or else raise Constraint_Error) and then
                    (Meaningful_For (Container, Parent)
                      or else raise Program_Error),
           Post => Node_Count (Container) =
                   Node_Count (Container)'Old + Count;
```

52/5 {*AI12-0112-1*}
```
procedure Append_Child (Container : in out Tree;
                        Parent    : in     Cursor;
                        New_Item  : in     Element_Type;
                        Count     : in     Count_Type := 1)
      with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                      or else raise Program_Error) and then
                    (Parent /= No_Element
                      or else raise Constraint_Error) and then
                    (Meaningful_For (Container, Parent)
                      or else raise Program_Error),
           Post => Node_Count (Container) =
                   Node_Count (Container)'Old + Count;
```

53/5 {*AI12-0112-1*}
```
procedure Delete_Children (Container : in out Tree;
                           Parent    : in     Cursor)
      with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                      or else raise Program_Error) and then
                    (Parent /= No_Element
                       or else raise Constraint_Error) and then
                    (Meaningful_For (Container, Parent)
                      or else raise Program_Error),
           Post => (Node_Count (Container) = Node_Count (Container)'Old -
                      Child_Count (Container, Parent)'Old) and then
                    Child_Count (Container, Parent) = 0;
```

54/5 {*AI12-0112-1*}
```
procedure Copy_Subtree (Target   : in out Tree;
                        Parent   : in     Cursor;
                        Before   : in     Cursor;
                        Source   : in     Cursor)
      with Pre  => (not Tampering_With_Cursors_Prohibited (Target)
                      or else raise Program_Error) and then
                    (Parent /= No_Element
                      or else raise Constraint_Error) and then
                    (Meaningful_For (Target, Parent)
                      or else raise Program_Error) and then
                    (Meaningful_For (Target, Before)
                      or else raise Program_Error) and then
                    (Before = No_Element or else
                      Target.Parent (Before) = Parent
                      or else raise Constraint_Error) and then
                    (not Is_Root (Source)
                      or else raise Constraint_Error),
           Post => Node_Count (Target) =
                   Node_Count (Target)'Old + Subtree_Node_Count (Source),
           Global => in all;
```

```
{AI12-0112-1}    procedure Copy_Local_Subtree (Target   : in out Tree;                54.1/5
                                               Parent   : in     Cursor;
                                               Before   : in     Cursor;
                                               Source   : in     Cursor)
      with Pre  => (not Tampering_With_Cursors_Prohibited (Target)
                       or else raise Program_Error) and then
                    (Parent /= No_Element
                       or else raise Constraint_Error) and then
                    (Meaningful_For (Target, Parent)
                       or else raise Program_Error) and then
                    (Meaningful_For (Target, Before)
                       or else raise Program_Error) and then
                    (Before = No_Element or else
                       Target.Parent (Before) = Parent
                       or else raise Constraint_Error) and then
                    (Meaningful_For (Target, Source)
                       or else raise Program_Error) and then
                    (not Is_Root (Source)
                       or else raise Constraint_Error),
           Post => Node_Count (Target) = Node_Count (Target)'Old +
                     Subtree_Node_Count (Target, Source);
{AI12-0112-1}    procedure Copy_Subtree (Target   : in out Tree;                      54.2/5
                                         Parent   : in     Cursor;
                                         Before   : in     Cursor;
                                         Source   : in     Tree;
                                         Subtree  : in     Cursor)
      with Pre  => (not Tampering_With_Cursors_Prohibited (Target)
                       or else raise Program_Error) and then
                    (Parent /= No_Element
                       or else raise Constraint_Error) and then
                    (Meaningful_For (Target, Parent)
                       or else raise Program_Error) and then
                    (Meaningful_For (Target, Before)
                       or else raise Program_Error) and then
                    (Before = No_Element or else
                       Target.Parent (Before) = Parent
                       or else raise Constraint_Error) and then
                    (Meaningful_For (Source, Subtree)
                       or else raise Program_Error) and then
                    (not Is_Root (Source, Subtree)
                       or else raise Constraint_Error),
           Post => Node_Count (Target) = Node_Count (Target)'Old +
                     Subtree_Node_Count (Source, Subtree);
```

55/5 {*AI12-0112-1*}
```
procedure Splice_Subtree (Target   : in out Tree;
                          Parent   : in     Cursor;
                          Before   : in     Cursor;
                          Source   : in out Tree;
                          Position : in out Cursor)
     with Pre  => (not Tampering_With_Cursors_Prohibited (Target)
                      or else raise Program_Error) and then
                  (not Tampering_With_Cursors_Prohibited (Source)
                      or else raise Program_Error) and then
                  (Parent /= No_Element
                      or else raise Constraint_Error) and then
                  (Meaningful_For (Target, Parent)
                      or else raise Program_Error) and then
                  (Meaningful_For (Target, Before)
                      or else raise Program_Error) and then
                  (Before = No_Element or else
                   Target.Parent (Before) /= Parent
                      or else raise Constraint_Error) and then
                  (Position /= No_Element
                      or else raise Constraint_Error) and then
                  (Has_Element (Source, Position)
                      or else raise Program_Error) and then
                  (Target'Has_Same_Storage (Source) or else
                   Position = Before or else
                   Is_Ancestor_Of (Target, Position, Parent)
                      or else raise Constraint_Error),
          Post => (declare
                      Org_Sub_Count renames
                         Subtree_Node_Count (Source, Position)'Old;
                      Org_Target_Count renames Node_Count (Target)'Old;
                   begin
                     (if not Target'Has_Same_Storage (Source) then
                         Node_Count (Target) = Org_Target_Count +
                            Org_Sub_Count and then
                         Node_Count (Source) = Node_Count (Source)'Old -
                            Org_Sub_Count and then
                         Has_Element (Target, Position)
                      else
                         Target.Parent (Position) = Parent and then
                         Node_Count (Target) = Org_Target_Count));
```

56/5 {*AI12-0112-1*}
```
procedure Splice_Subtree (Container: in out Tree;
                          Parent   : in     Cursor;
                          Before   : in     Cursor;
                          Position : in     Cursor)
     with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                      or else raise Program_Error) and then
                  (Parent /= No_Element
                      or else raise Constraint_Error) and then
                  (Meaningful_For (Container, Parent)
                     or else raise Program_Error) and then
                  (Meaningful_For (Container, Before)
                     or else raise Program_Error) and then
                  (Before = No_Element or else
                   Container.Parent (Before) /= Parent
                     or else raise Constraint_Error) and then
                  (Position /= No_Element
                     or else raise Constraint_Error) and then
                  (Has_Element (Container, Position)
                     or else raise Program_Error) and then
                  (Position = Before or else
                   Is_Ancestor_Of (Container, Position, Parent)
                     or else raise Constraint_Error),
          Post => (Node_Count (Container) =
                     Node_Count (Container)'Old and then
                   Container.Parent (Position) = Parent);
```

```
{AI12-0112-1}    procedure Splice_Children (Target          : in out Tree;                    57/5
                                            Target_Parent   : in     Cursor;
                                            Before          : in     Cursor;
                                            Source          : in out Tree;
                                            Source_Parent   : in     Cursor)
      with Pre  => (not Tampering_With_Cursors_Prohibited (Target)
                       or else raise Program_Error) and then
                    (not Tampering_With_Cursors_Prohibited (Source)
                       or else raise Program_Error) and then
                    (Target_Parent /= No_Element
                       or else raise Constraint_Error) and then
                    (Meaningful_For (Target, Target_Parent)
                       or else raise Program_Error) and then
                    (Meaningful_For (Target, Before)
                       or else raise Program_Error) and then
                    (Source_Parent /= No_Element
                       or else raise Constraint_Error) and then
                    (Meaningful_For (Source, Source_Parent)
                       or else raise Program_Error) and then
                    (Before = No_Element or else
                     Parent (Target, Before) /= Target_Parent
                       or else raise Constraint_Error) and then
                    (Target'Has_Same_Storage (Source) or else
                     Target_Parent = Source_Parent or else
                     Is_Ancestor_Of (Target, Source_Parent, Target_Parent)
                       or else raise Constraint_Error),
           Post => (declare
                       Org_Child_Count renames
                          Child_Count (Source, Source_Parent)'Old;
                       Org_Target_Count renames Node_Count (Target)'Old;
                    begin
                      (if not Target'Has_Same_Storage (Source) then
                          Node_Count (Target) = Org_Target_Count +
                             Org_Child_Count and then
                          Node_Count (Source) = Node_Count (Source)'Old -
                             Org_Child_Count
                       else
                          Node_Count (Target) = Org_Target_Count));

{AI12-0112-1}    procedure Splice_Children (Container       : in out Tree;                    58/5
                                            Target_Parent   : in     Cursor;
                                            Before          : in     Cursor;
                                            Source_Parent   : in     Cursor)
      with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                      or else raise Program_Error) and then
                    (Target_Parent /= No_Element
                      or else raise Constraint_Error) and then
                    (Meaningful_For (Container, Target_Parent)
                      or else raise Program_Error) and then
                    (Meaningful_For (Container, Before)
                      or else raise Program_Error) and then
                    (Source_Parent /= No_Element
                      or else raise Constraint_Error) and then
                    (Meaningful_For (Container, Source_Parent)
                      or else raise Program_Error) and then
                    (Before = No_Element or else
                     Parent (Container, Before) /= Target_Parent
                      or else raise Constraint_Error) and then
                    (Target_Parent = Source_Parent or else
                     Is_Ancestor_Of (Container, Source_Parent, Target_Parent)
                      or else raise Constraint_Error),
           Post => Node_Count (Container) = Node_Count (Container)'Old;

{AI12-0112-1}    function Parent (Position : Cursor) return Cursor                             59/5
      with Nonblocking, Global => in all, Use_Formal => null,
           Post => (if Position = No_Element or else
                       Is_Root (Position) then Parent'Result = No_Element);
```

59.1/5 {*AI12-0112-1*}
```
function Parent (Container : Tree;
                 Position  : Cursor) return Cursor
   with Nonblocking, Global => null, Use_Formal => null,
        Pre  => Meaningful_For (Container, Position)
                   or else raise Program_Error,
        Post => (if Position = No_Element or else
                   Is_Root (Container, Position)
                   then Parent'Result = No_Element
                 else Has_Element (Container, Parent'Result));
```

60/5 {*AI12-0112-1*}
```
function First_Child (Parent : Cursor) return Cursor
   with Nonblocking, Global => in all, Use_Formal => null,
        Pre  => Parent /= No_Element or else raise Constraint_Error;
```

60.1/5 {*AI12-0112-1*}
```
function First_Child (Container : Tree;
                      Parent    : Cursor) return Cursor
   with Nonblocking, Global => null, Use_Formal => null,
        Pre  => (Parent /= No_Element
                   or else raise Constraint_Error) and then
                 (Meaningful_For (Container, Parent)
                   or else raise Program_Error),
        Post => First_Child'Result = No_Element or else
                Has_Element (Container, First_Child'Result);
```

61/5 {*AI12-0112-1*}
```
function First_Child_Element (Parent : Cursor) return Element_Type
   with Nonblocking, Global => in all, Use_Formal => Element_Type,
        Pre  => (Parent /= No_Element and then
                 Last_Child (Parent) /= No_Element)
                    or else raise Constraint_Error;
```

61.1/5 {*AI12-0112-1*}
```
function First_Child_Element (Container : Tree;
                              Parent    : Cursor) return Element_Type
   with Nonblocking, Global => null, Use_Formal => Element_Type,
        Pre  => (Parent /= No_Element
                   or else raise Constraint_Error) and then
                 (Meaningful_For (Container, Parent)
                   or else raise Program_Error) and then
                 (First_Child (Container, Parent) /= No_Element
                   or else raise Constraint_Error);
```

62/5 {*AI12-0112-1*}
```
function Last_Child (Parent : Cursor) return Cursor
   with Nonblocking, Global => in all, Use_Formal => null,
        Pre  => Parent /= No_Element or else raise Constraint_Error;
```

62.1/5 {*AI12-0112-1*}
```
function Last_Child (Container : Tree;
                     Parent    : Cursor) return Cursor
   with Nonblocking, Global => null, Use_Formal => null,
        Pre  => (Parent /= No_Element
                   or else raise Constraint_Error) and then
                 (Meaningful_For (Container, Parent)
                   or else raise Program_Error),
        Post => Last_Child'Result = No_Element or else
                Has_Element (Container, Last_Child'Result);
```

63/5 {*AI12-0112-1*}
```
function Last_Child_Element (Parent : Cursor) return Element_Type
   with Nonblocking, Global => in all, Use_Formal => Element_Type,
        Pre  => (Parent /= No_Element and then
                 Last_Child (Parent) /= No_Element)
                    or else raise Constraint_Error;
```

63.1/5 {*AI12-0112-1*}
```
function Last_Child_Element (Container : Tree;
                             Parent    : Cursor) return Element_Type
   with Nonblocking, Global => null, Use_Formal => Element_Type,
        Pre  => (Parent /= No_Element
                   or else raise Constraint_Error) and then
                 (Meaningful_For (Container, Parent)
                   or else raise Program_Error) and then
                 (Last_Child (Container, Parent) /= No_Element
                   or else raise Constraint_Error);
```

```
{AI12-0112-1}    function Next_Sibling (Position : Cursor) return Cursor                64/5
      with Nonblocking, Global => in all, Use_Formal => null,
           Post => (if Position = No_Element
                       then Next_Sibling'Result = No_Element);

{AI12-0112-1}    function Next_Sibling (Container : Tree;                               64.1/5
                                        Position  : Cursor) return Cursor
      with Nonblocking, Global => null, Use_Formal => null,
           Pre  => Meaningful_For (Container, Position)
                      or else raise Program_Error,
           Post => (if Next_Sibling'Result = No_Element then
                      Position = No_Element or else
                      Is_Root (Container, Position) or else
                      Last_Child (Container, Parent (Container, Position))
                         = Position
                    else Has_Element (Container, Next_Sibling'Result));

{AI12-0112-1}    procedure Next_Sibling (Position : in out Cursor)                      65/5
      with Nonblocking, Global => in all, Use_Formal => null;

{AI12-0112-1}    procedure Next_Sibling (Container : in     Tree;                       65.1/5
                                         Position  : in out Cursor)
      with Nonblocking, Global => null, Use_Formal => null,
           Pre  => Meaningful_For (Container, Position)
                      or else raise Program_Error,
           Post => (if Position /= No_Element
                    then Has_Element (Container, Position));

{AI12-0112-1}    function Previous_Sibling (Position : Cursor) return Cursor            66/5
      with Nonblocking, Global => in all, Use_Formal => null,
           Post => (if Position = No_Element
                       then Previous_Sibling'Result = No_Element);

{AI12-0112-1}    function Previous_Sibling (Container : Tree;                           66.1/5
                                            Position  : Cursor) return Cursor
      with Nonblocking, Global => null, Use_Formal => null,
           Pre  => Meaningful_For (Container, Position)
                      or else raise Program_Error,
           Post => (if Previous_Sibling'Result = No_Element then
                      Position = No_Element or else
                      Is_Root (Container, Position) or else
                      First_Child (Container, Parent (Container, Position))
                         = Position
                    else Has_Element (Container, Previous_Sibling'Result));

{AI12-0112-1}    procedure Previous_Sibling (Position : in out Cursor)                  67/5
      with Nonblocking, Global => in all, Use_Formal => null;

{AI12-0112-1}    procedure Previous_Sibling (Container : in     Tree;                   67.1/5
                                            Position  : in out Cursor)
      with Nonblocking, Global => null, Use_Formal => null,
           Pre  => Meaningful_For (Container, Position)
                      or else raise Program_Error,
           Post => (if Position /= No_Element
                    then Has_Element (Container, Position));

{AI05-0136-1} {AI05-0248-1} {AI12-0112-1}    procedure Iterate_Children               68/5
        (Parent  : in Cursor;
         Process : not null access procedure (Position : in Cursor))
      with Allows_Exit,
           Pre    => Parent /= No_Element or else raise Constraint_Error,
           Global => in all, Use_Formal => null;

{AI12-0112-1}    procedure Iterate_Children                                             68.1/5
        (Container : in Tree;
         Parent    : in Cursor;
         Process   : not null access procedure (Position : in Cursor))
      with Allows_Exit,
           Pre  => (Parent /= No_Element
                      or else raise Constraint_Error) and then
                   (Meaningful_For (Container, Parent)
                      or else raise Program_Error);
```

69/5 {*AI05-0136-1*} {*AI05-0248-1*} {*AI12-0112-1*}
```
   procedure Reverse_Iterate_Children
      (Parent  : in Cursor;
       Process : not null access procedure (Position : in Cursor))
   with Allows_Exit,
        Pre    => Parent /= No_Element or else raise Constraint_Error,
        Global => in all, Use_Formal => null;
```

69.1/5 {*AI12-0112-1*}
```
   procedure Reverse_Iterate_Children
      (Container : in Tree;
       Parent    : in Cursor;
       Process   : not null access procedure (Position : in Cursor))
   with Allows_Exit,
        Pre  => (Parent /= No_Element
                    or else raise Constraint_Error) and then
                (Meaningful_For (Container, Parent)
                    or else raise Program_Error);
```

70/5 {*AI05-0212-1*} {*AI12-0112-1*} {*AI12-0266-1*}
```
   function Iterate_Children (Container : in Tree; Parent : in Cursor)
      return Tree_Iterator_Interfaces.Parallel_Reversible_Iterator'Class
      with Pre  => (Parent /= No_Element
                    or else raise Constraint_Error) and then
                   (Meaningful_For (Container, Parent)
                    or else raise Program_Error),
           Post => Tampering_With_Cursors_Prohibited (Container);
```

70.1/5 {*AI12-0111-1*}
```
   package Stable is
```

70.2/5 {*AI12-0111-1*} {*AI12-0399-1*}
```
      type Tree (Base : not null access Multiway_Trees.Tree) is
         tagged limited private
         with Constant_Indexing => Constant_Reference,
              Variable_Indexing => Reference,
              Default_Iterator  => Iterate,
              Iterator_Element  => Element_Type,
              Stable_Properties => (Node_Count),
              Global            => null,
              Default_Initial_Condition => Node_Count (Tree) = 1,
              Preelaborable_Initialization;
```

70.3/5 {*AI12-0111-1*} {*AI12-0399-1*}
```
      type Cursor is private
         with Preelaborable_Initialization;
```

70.4/5 {*AI12-0111-1*}
```
      Empty_Tree : constant Tree;
```

70.5/5 {*AI12-0111-1*}
```
      No_Element : constant Cursor;
```

70.6/5 {*AI12-0111-1*}
```
      function Has_Element (Position : Cursor) return Boolean
         with Nonblocking, Global => in all, Use_Formal => null;
```

70.7/5 {*AI12-0111-1*}
```
      package Tree_Iterator_Interfaces is new
         Ada.Iterator_Interfaces (Cursor, Has_Element);
```

70.8/5 {*AI12-0111-1*}
```
      procedure Assign (Target : in out Multiway_Trees.Tree;
                        Source : in Tree)
         with Post => Node_Count (Source) = Node_Count (Target);
```

70.9/5 {*AI12-0111-1*}
```
      function Copy (Source : Multiway_Trees.Tree) return Tree
         with Post => Node_Count (Copy'Result) = Node_Count (Source);
```

70.10/5 {*AI12-0111-1*}
```
      type Constant_Reference_Type
         (Element : not null access constant Element_Type) is private
         with Implicit_Dereference => Element,
              Nonblocking, Global => null,
              Default_Initial_Condition => (raise Program_Error);
```

70.11/5 {*AI12-0111-1*}
```
      type Reference_Type
         (Element : not null access Element_Type) is private
         with Implicit_Dereference => Element,
              Nonblocking, Global => null,
              Default_Initial_Condition => (raise Program_Error);
```

70.12/5 {*AI12-0111-1*}
```
      -- Additional subprograms as described in the text
      -- are declared here.
```

```
{AI12-0111-1}     private                                                    70.13/5
{AI12-0111-1}        ... -- not specified by the language                    70.14/5
{AI12-0111-1}     end Stable;                                                70.15/5
private                                                                      71/3
   ... -- not specified by the language
end Ada.Containers.Multiway_Trees;
```

{*AI05-0136-1*} The actual function for the generic formal function "=" on Element_Type values is expected to define a reflexive and symmetric relationship and return the same result value each time it is called with a particular pair of values. If it behaves in some other manner, the functions Find, Reverse_Find, Equal_Subtree, and "=" on tree values return an unspecified value. The exact arguments and number of calls of this generic formal function by the functions Find, Reverse_Find, Equal_Subtree, and "=" on tree values are unspecified. 72/3

{*AI05-0136-1*} The type Tree is used to represent trees. The type Tree needs finalization (see 7.6). 73/3

{*AI05-0136-1*} {*AI05-0248-1*} Empty_Tree represents the empty Tree object. It contains only the root node (Node_Count (Empty_Tree) returns 1). If an object of type Tree is not otherwise initialized, it is initialized to the same value as Empty_Tree. 74/3

{*AI05-0136-1*} No_Element represents a cursor that designates no element. If an object of type Cursor is not otherwise initialized, it is initialized to the same value as No_Element. 75/3

{*AI05-0136-1*} {*AI12-0434-1*} The primitive "=" operator for type Cursor returns True if both cursors are No_Element, or designate the same element in the same container. 76/5

To be honest: {*AI12-0434-1*} “The primitive "=" operator” is the one with two parameters of type Cursor which returns Boolean. We're not talking about some other (hidden) primitive function named "=". 76.a/5

{*AI05-0136-1*} Execution of the default implementation of the Input, Output, Read, or Write attribute of type Cursor raises Program_Error. 77/3

{*AI05-0136-1*} {*AI05-0248-1*} {*AI05-0262-1*} {*AI12-0437-1*} Tree'Write for a Tree object *T* writes Node_Count(*T*) - 1 elements of the tree to the stream. It may also write additional information about the tree. 78/5

{*AI05-0136-1*} {*AI05-0248-1*} {*AI05-0262-1*} Tree'Read reads the representation of a tree from the stream, and assigns to *Item* a tree with the same elements and structure as was written by Tree'Write. 79/3

Ramification: Streaming more elements than the container holds is wrong. For implementation implications of this rule, see the Implementation Note in A.18.2. 79.a/3

{*AI05-0136-1*} {*AI12-0111-1*} {*AI12-0112-1*} [Some operations check for “tampering with cursors” of a container because they depend on the set of elements of the container remaining constant, and others check for “tampering with elements” of a container because they depend on elements of the container not being replaced.] When tampering with cursors is *prohibited* for a particular tree object *T*, Program_Error is propagated by the finalization of *T*[, as well as by a call that passes *T* to certain of the operations of this package, as indicated by the precondition of such an operation]. Similarly, when tampering with elements is *prohibited* for *T*, Program_Error is propagated by a call that passes *T* to certain of the other operations of this package, as indicated by the precondition of such an operation. 80/5

Paragraphs 81 through 90 are removed as preconditions now describe these rules.

Ramification: We don't need to explicitly mention assignment_statement, because that finalizes the target object as part of the operation, and finalization of an object is already defined as tampering with cursors. 85.a.1/3

91/5 {*AI12-0112-1*}
```
function Has_Element (Position : Cursor) return Boolean
   with Nonblocking, Global => in all, Use_Formal => null;
```

92/3 Returns True if Position designates an element, and returns False otherwise. [In particular, Has_Element returns False if the cursor designates a root node or equals No_Element.]

92.a/3 **To be honest:** {*AI05-0005-1*} {*AI05-0136-1*} This function might not detect cursors that designate deleted elements; such cursors are invalid (see below) and the result of calling Has_Element with an invalid cursor is unspecified (but not erroneous).

92.1/5
```
function Has_Element (Container : Tree; Position : Cursor)
   return Boolean
   with Nonblocking, Global => null, Use_Formal => null;
```

92.2/5 {*AI12-0112-1*} Returns True if Position designates an element in Container, and returns False otherwise. [In particular, Has_Element returns False if the cursor designates a root node or equals No_Element.]

93/3
```
function Equal_Subtree (Left_Position : Cursor;
                        Right_Position: Cursor) return Boolean;
```

94/3 {*AI05-0136-1*} {*AI05-0248-1*} {*AI05-0262-1*} {*AI05-0264-1*} If Left_Position or Right_Position equals No_Element, propagates Constraint_Error. If the number of child nodes of the element designated by Left_Position is different from the number of child nodes of the element designated by Right_Position, the function returns False. If Left_Position designates a root node and Right_Position does not, the function returns False. If Right_Position designates a root node and Left_Position does not, the function returns False. Unless both cursors designate a root node, the elements are compared using the generic formal equality operator. If the result of the element comparison is False, the function returns False. Otherwise, it calls Equal_Subtree on a cursor designating each child element of the element designated by Left_Position and a cursor designating the corresponding child element of the element designated by Right_Position. If any such call returns False, the function returns False; otherwise, it returns True. Any exception raised during the evaluation of element equality is propagated.

94.a/3 **Ramification:** Left_Position and Right_Position do not need to be from the same tree.

94.b/3 **Implementation Note:** This wording describes the canonical semantics. However, the order and number of calls on the formal equality function is unspecified for all of the operations that use it in this package, so an implementation can call it as many or as few times as it needs to get the correct answer. Similarly, a global rule (see the introduction of Annex A) says that language-defined routines are not affected by overriding of other language-defined routines. This means that no reasonable program can tell how many times Equal_Subtree is called, and thus an implementation can call it as many or as few times as it needs to get the correct answer. Specifically, there is no requirement to call the formal equality or Equal_Subtree additional times once the answer has been determined.

95/3
```
function "=" (Left, Right : Tree) return Boolean;
```

96/3 {*AI05-0136-1*} {*AI05-0262-1*} If Left and Right denote the same tree object, then the function returns True. Otherwise, it calls Equal_Subtree with cursors designating the root nodes of Left and Right; the result is returned. Any exception raised during the evaluation of Equal_Subtree is propagated.

96.a/3 **Implementation Note:** Similar considerations apply here as apply to Equal_Subtree. The actual number of calls performed is unspecified.

96.1/5
```
function Tampering_With_Cursors_Prohibited
   (Container : Tree) return Boolean
   with Nonblocking, Global => null, Use_Formal => null;
```

96.2/5 {*AI12-0112-1*} Returns True if tampering with cursors or tampering with elements is currently prohibited for Container, and returns False otherwise.

96.b/5 **Reason:** {*AI12-0112-1*} Prohibiting tampering with elements also needs to prohibit tampering with cursors, as deleting an element is similar to replacing it.

Implementation Note: {*AI12-0112-1*} Various contracts elsewhere in this specification require that this function be implemented with synchronized data. Moreover, it is possible for tampering to be prohibited by multiple operations (sequentially or in parallel). Therefore, tampering needs to be implemented with an atomic or protected counter. The counter is initialized to zero, and is incremented when tampering is prohibited, and decremented when leaving an area that prohibited tampering. Function Tampering_With_Cursors_Prohibited returns True if the counter is nonzero. (Note that any case where the result is not well-defined for one task is incorrect use of shared variables and would be erroneous by the rules of 9.10, so no special protection is needed to read the counter.) 96.c/5

```
function Tampering_With_Elements_Prohibited                                    96.3/5
   (Container : Tree) return Boolean
   with Nonblocking, Global => null, Use_Formal => null;
```

{*AI12-0112-1*} Always returns False[, regardless of whether tampering with elements is prohibited]. 96.4/5

Reason: {*AI12-0111-1*} A definite element cannot change size, so we allow operations that tamper with elements even when tampering with elements is prohibited. That's not true for the indefinite containers, which is why this kind of tampering exists. 96.d/5

```
function Is_Empty (Container : Tree) return Boolean                            96.5/5
   with Nonblocking, Global => null, Use_Formal => null,
        Post => Is_Empty'Result = (Node_Count (Container) = 1);
```

{*AI05-0136-1*} {*AI12-0112-1*} Returns True if Container is empty. 96.6/5

Ramification: An empty tree contains just the root node. 96.e/5

```
{AI12-0112-1} function Node_Count (Container : Tree) return Count_Type         97/5
   with Nonblocking, Global => null, Use_Formal => null;
```

{*AI05-0136-1*} Node_Count returns the number of nodes in Container. 98/3

Ramification: Since all tree objects have a root node, this can never return a value of 0. Node_Count (Some_Tree) should always equal Subtree_Node_Count (Root (Some_Tree)). 98.a/3

```
{AI12-0112-1} function Subtree_Node_Count (Position : Cursor) return Count_Type  99/5
   with Nonblocking, Global => in all, Use_Formal => null);
```

{*AI05-0136-1*} {*AI05-0248-1*} If Position is No_Element, Subtree_Node_Count returns 0; otherwise, Subtree_Node_Count returns the number of nodes in the subtree that is rooted by Position. 100/3

```
function Subtree_Node_Count (Container : Tree; Position : Cursor)              101/5
   return Count_Type
   with Pre => Meaningful_For (Container, Position)
                  or else raise Program_Error,
        Nonblocking, Global => null, Use_Formal => null;
```

{*AI05-0136-1*} {*AI12-0112-1*} If Position is No_Element, Subtree_Node_Count returns 0; otherwise, Subtree_Node_Count returns the number of nodes in the subtree of Container that is rooted by Position. 102/5

This paragraph was deleted.{*AI12-0112-1*} 102.a/5

Reason: {*AI12-0112-1*} We raise Program_Error if Position belongs to some other container because we have promised to read only the container passed to this function. Determining the answer requires reading the container that Position belongs to, which we've promised not to do if it is not Container. We don't make this check for functions like Has_Element and Is_Root which do not require reading another container to determine the answer, but we do make it for most functions. 102.b/5

```
{AI12-0112-1} function Depth (Position : Cursor) return Count_Type             103/5
   with Nonblocking, Global => in all, Use_Formal => null;
```

{*AI05-0136-1*} {*AI05-0248-1*} If Position equals No_Element, Depth returns 0; otherwise, Depth returns the number of ancestor nodes of the node designated by Position (including the node itself). 104/3

Ramification: Depth (Root (Some_Tree)) = 1. 104.a/3

104.1/5
```
function Depth (Container : Tree; Position : Cursor)
   return Count_Type
   with Pre => Meaningful_For (Container, Position)
                  or else raise Program_Error,
        Nonblocking, Global => null, Use_Formal => null;
```

104.2/5 {*AI12-0112-1*} If Position equals No_Element, Depth returns 0; otherwise, Depth returns the number of ancestor nodes of the node of Container designated by Position (including the node itself).

105/5
```
{AI12-0112-1} function Is_Root (Position : Cursor) return Boolean
   with Nonblocking, Global => in all, Use_Formal => null;
```

106/3 {*AI05-0136-1*} {*AI05-0248-1*} Is_Root returns True if the Position designates the root node of some tree; and returns False otherwise.

106.1/5
```
function Is_Root (Container : Tree; Position : Cursor)
   return Boolean
   with Nonblocking, Global => null, Use_Formal => null;
```

106.2/5 {*AI12-0112-1*} Is_Root returns True if the Position designates the root node of Container; and returns False otherwise.

106.a/5 **Ramification:** The two parameter Is_Root returns False even if Position is the root of some other tree than Container.

107/5
```
{AI12-0112-1} function Is_Leaf (Position : Cursor) return Boolean
   with Nonblocking, Global => in all, Use_Formal => null;
```

108/3 {*AI05-0136-1*} Is_Leaf returns True if Position designates a node that does not have any child nodes; and returns False otherwise.

108.a/3 **Ramification:** Is_Leaf returns False if passed No_Element, since No_Element does not designate a node. Is_Leaf can be passed a cursor that designates the root node; Is_Leaf will return True if passed the root node of an empty tree.

108.1/5
```
function Is_Leaf (Container : Tree; Position : Cursor)
   return Boolean
   with Pre => Meaningful_For (Container, Position)
                  or else raise Program_Error,
        Nonblocking, Global => null, Use_Formal => null;
```

108.2/5 {*AI12-0112-1*} Is_Leaf returns True if Position designates a node in Container that does not have any child nodes; and returns False otherwise.

108.3/5
```
function Is_Ancestor_Of (Container : Tree;
                         Parent   : Cursor;
                         Position : Cursor) return Boolean
   with Pre => (Meaningful_For (Container, Position)
                  or else raise Program_Error) and then
               (Meaningful_For (Container, Parent)
                  or else raise Program_Error),
        Nonblocking, Global => null, Use_Formal => null;
```

108.4/5 {*AI12-0112-1*} Is_Ancestor_Of returns True if Parent designates an ancestor node of Position (including Position itself), and returns False otherwise.

109/5
```
{AI12-0112-1} function Root (Container : Tree) return Cursor
   with Nonblocking, Global => null, Use_Formal => null,
        Post => Root'Result /= No_Element and then
                not Has_Element (Container, Root'Result);
```

110/3 {*AI05-0136-1*} Root returns a cursor that designates the root node of Container.

110.a/3 **Ramification:** There is always a root node, even in an empty container, so this function never returns No_Element.

```
{AI12-0112-1} procedure Clear (Container : in out Tree)                                    111/5
   with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error,
        Post => Node_Count (Container) = 1;
```

{*AI05-0136-1*} Removes all the elements from Container. 112/3

Ramification: The root node is not removed; all trees have a root node. 112.a/3

```
function Element (Position : Cursor) return Element_Type                                   113/5
   with Pre => (Position /= No_Element or else
                   raise Constraint_Error) and then
               (Has_Element (Position) or else raise Program_Error),
        Nonblocking, Global => in all, Use_Formal => Element_Type;
```

{*AI05-0136-1*} {*AI12-0112-1*} Element returns the element designated by Position. 114/5

Ramification: The root node does not contain an element, so that value cannot be read or written. 114.a/3

```
function Element (Container : Tree;                                                        114.1/5
                  Position  : Cursor) return Element_Type
   with Pre => (Position /= No_Element
                   or else raise Constraint_Error) and then
               (Has_Element (Container, Position)
                   or else raise Program_Error),
        Nonblocking, Global => null, Use_Formal => Element_Type;
```

{*AI12-0112-1*} Element returns the element designated by Position in Container. 114.2/5

```
procedure Replace_Element (Container : in out Tree;                                        115/5
                           Position  : in     Cursor;
                           New_item  : in     Element_Type)
   with Pre  => (not Tampering_With_Elements_Prohibited (Container)
                   or else raise Program_Error) and then
                (Position /= No_Element
                   or else raise Constraint_Error) and then
                (Has_Element (Container, Position)
                   or else raise Program_Error);
```

{*AI05-0136-1*} {*AI05-0264-1*} {*AI12-0112-1*} {*AI12-0196-1*} Replace_Element assigns the value New_Item to the element designated by Position. For the purposes of determining whether the parameters overlap in a call to Replace_Element, the Container parameter is not considered to overlap with any object [(including itself)]. 116/5

```
procedure Query_Element                                                                    117/5
  (Position : in Cursor;
   Process  : not null access procedure (Element : in Element_Type))
   with Pre  => (Position /= No_Element
                   or else raise Constraint_Error) and then
                (Has_Element (Position)
                   or else raise Program_Error),
     Global => in all;
```

{*AI05-0136-1*} {*AI05-0265-1*} {*AI12-0112-1*} Query_Element calls Process.**all** with the element designated by Position as the argument. Tampering with the elements of the tree that contains the element designated by Position is prohibited during the execution of the call on Process.**all**. Any exception raised by Process.**all** is propagated. 118/5

```
procedure Query_Element                                                                    118.1/5
  (Container : in Tree;
   Position  : in Cursor;
   Process   : not null access procedure (Element : in Element_Type))
   with Pre  => (Position /= No_Element
                   or else raise Constraint_Error) and then
                (Has_Element (Container, Position)
                   or else raise Program_Error);
```

118.2/5 {*AI12-0112-1*} Query_Element calls Process.**all** with the element designated by Position as the argument. Tampering with the elements of Container is prohibited during the execution of the call on Process.**all**. Any exception raised by Process.**all** is propagated.

119/5
```
procedure Update_Element
   (Container : in out Tree;
    Position  : in     Cursor;
    Process   : not null access procedure
                     (Element : in out Element_Type))
    with Pre  => (Position /= No_Element
                     or else raise Constraint_Error) and then
                  (Has_Element (Container, Position)
                     or else raise Program_Error);
```

120/5 {*AI05-0136-1*} {*AI05-0264-1*} {*AI05-0265-1*} {*AI12-0112-1*} Update_Element calls Process.**all** with the element designated by Position as the argument. Tampering with the elements of Container is prohibited during the execution of the call on Process.**all**. Any exception raised by Process.**all** is propagated.

121/3 If Element_Type is unconstrained and definite, then the actual Element parameter of Process.**all** shall be unconstrained.

121.a/3 **Ramification:** This means that the elements cannot be directly allocated from the heap; it must be possible to change the discriminants of the element in place.

122/5
```
{AI12-0112-1} type Constant_Reference_Type
      (Element : not null access constant Element_Type) is private
   with Implicit_Dereference => Element,
        Nonblocking, Global => in out synchronized,
        Default_Initial_Condition => (raise Program_Error);
```

123/5
```
{AI12-0112-1} type Reference_Type (Element : not null access Element_Type) is
private
   with Implicit_Dereference => Element,
        Nonblocking, Global => in out synchronized,
        Default_Initial_Condition => (raise Program_Error);
```

124/3 {*AI05-0212-1*} The types Constant_Reference_Type and Reference_Type need finalization.

125/5 *This paragraph was deleted.*{*AI12-0112-1*}

125.a/3 **Reason:** It is expected that Reference_Type (and Constant_Reference_Type) will be a controlled type, for which finalization will have some action to terminate the tampering check for the associated container. If the object is created by default, however, there is no associated container. Since this is useless, and supporting this case would take extra work, we define it to raise an exception.

126/5
```
{AI12-0112-1} function Constant_Reference (Container : aliased in Tree;
                                Position  : in Cursor)
   return Constant_Reference_Type
   with Pre  => (Position /= No_Element
                    or else raise Constraint_Error) and then
                 (Has_Element (Container, Position)
                    or else raise Program_Error),
        Post   => Tampering_With_Cursors_Prohibited (Container),
        Nonblocking, Global => null, Use_Formal => null;
```

127/3 {*AI05-0212-1*} {*AI05-0269-1*} This function (combined with the Constant_Indexing and Implicit_Dereference aspects) provides a convenient way to gain read access to an individual element of a tree given a cursor.

128/5 {*AI05-0212-1*} {*AI05-0265-1*} {*AI12-0112-1*} Constant_Reference returns an object whose discriminant is an access value that designates the element designated by Position. Tampering with the elements of Container is prohibited while the object returned by Constant_Reference exists and has not been finalized.

```
{AI12-0112-1} function Reference (Container : aliased in out Tree;                    129/5
                                  Position  : in Cursor)
   return Reference_Type
   with Pre  => (Position /= No_Element
                    or else raise Constraint_Error) and then
                 (Has_Element (Container, Position)
                    or else raise Program_Error),
        Post   => Tampering_With_Cursors_Prohibited (Container),
        Nonblocking, Global => null, Use_Formal => null;
```

{*AI05-0212-1*} {*AI05-0269-1*} This function (combined with the Variable_Indexing and Implicit_Dereference aspects) provides a convenient way to gain read and write access to an individual element of a tree given a cursor. 130/3

{*AI05-0212-1*} {*AI05-0265-1*} {*AI12-0112-1*} Reference returns an object whose discriminant is an access value that designates the element designated by Position. Tampering with the elements of Container is prohibited while the object returned by Reference exists and has not been finalized. 131/5

```
{AI12-0112-1} procedure Assign (Target : in out Tree; Source : in Tree)               132/5
   with Pre  => not Tampering_With_Cursors_Prohibited (Target)
                    or else raise Program_Error,
        Post => Node_Count (Source) = Node_Count (Target);
```

{*AI05-0136-1*} {*AI05-0248-1*} If Target denotes the same object as Source, the operation has no effect. Otherwise, the elements of Source are copied to Target as for an assignment_statement assigning Source to Target. 133/3

Ramification: Each element in Target has a parent element that corresponds to the parent element of the Source element, and has child elements that correspond to the child elements of the Source element. 133.a/3

Discussion: {*AI05-0005-1*} This routine exists for compatibility with the bounded tree container. For an unbounded tree, `Assign(A, B)` and `A := B` behave identically. For a bounded tree, := will raise an exception if the container capacities are different, while Assign will not raise an exception if there is enough room in the target. 133.b/3

```
{AI12-0112-1} function Copy (Source : Tree) return Tree                               134/5
   with Post =>
           Node_Count (Copy'Result) = Node_Count (Source) and then
           not Tampering_With_Elements_Prohibited (Copy'Result) and then
           not Tampering_With_Cursors_Prohibited (Copy'Result);
```

{*AI05-0136-1*} Returns a tree with the same structure as Source and whose elements are initialized from the corresponding elements of Source. 135/3

```
{AI12-0112-1} procedure Move (Target : in out Tree;                                   136/5
                              Source : in out Tree)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Target)
                    or else raise Program_Error) and then
                 (not Tampering_With_Cursors_Prohibited (Source)
                    or else raise Program_Error),
        Post => (if not Target'Has_Same_Storage (Source) then
                    Node_Count (Target) = Node_Count (Source'Old) and then
                    Node_Count (Source) = 1);
```

{*AI05-0136-1*} {*AI05-0248-1*} If Target denotes the same object as Source, then the operation has no effect. Otherwise, Move first calls Clear (Target). Then, the nodes other than the root node in Source are moved to Target (in the same positions). After Move completes, Node_Count (Target) is the number of nodes originally in Source, and Node_Count (Source) is 1. 137/3

138/5
```
procedure Delete_Leaf (Container : in out Tree;
                       Position  : in out Cursor)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error) and then
                 (Position /= No_Element
                    or else raise Constraint_Error) and then
                 (Has_Element (Container, Position)
                    or else raise Program_Error) and then
                 (Is_Leaf (Container, Position)
                    or else raise Constraint_Error),
        Post =>
           Node_Count (Container)'Old = Node_Count (Container) + 1 and then
           Position = No_Element;
```

139/5 {*AI05-0136-1*} {*AI05-0248-1*} {*AI12-0112-1*} Delete_Leaf removes (from Container) the element designated by Position, and Position is set to No_Element.

139.a/3 **Ramification:** The check on Position checks that the cursor does not belong to some other Container. This check implies that a reference to the container is included in the cursor value. This wording is not meant to require detection of dangling cursors; such cursors are defined to be invalid, which means that execution is erroneous, and any result is allowed (including not raising an exception).

139.b/3 The root node cannot be deleted.

140/5
```
procedure Delete_Subtree (Container : in out Tree;
                          Position  : in out Cursor)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error) and then
                 (Position /= No_Element
                    or else raise Constraint_Error) and then
                 (Has_Element (Container, Position)
                    or else raise Program_Error),
        Post => Node_Count (Container)'Old = Node_Count (Container) +
                    Subtree_Node_Count (Container, Position)'Old and then
                 Position = No_Element;
```

141/5 {*AI05-0136-1*} {*AI05-0264-1*} {*AI05-0269-1*} {*AI12-0112-1*} Delete_Subtree removes (from Container) the subtree designated by Position (that is, all descendants of the node designated by Position including the node itself), and Position is set to No_Element.

141.a/3 **Ramification:** The root node cannot be deleted. To delete the entire contents of the tree, call Clear(Container).

142/5
```
procedure Swap (Container : in out Tree;
                I, J      : in     Cursor)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error) and then
                 (I /= No_Element or else Constraint_Error) and then
                 (J /= No_Element or else Constraint_Error) and then
                 (Has_Element (Container, I)
                    or else raise Program_Error) and then
                 (Has_Element (Container, J)
                    or else raise Program_Error);
```

143/5 {*AI05-0136-1*} {*AI12-0112-1*} Swap exchanges the values of the elements designated by I and J.

143.a/3 **Ramification:** After a call to Swap, I designates the element value previously designated by J, and J designates the element value previously designated by I. The position of the elements do not change; for instance, the parent node and the first child node of I are unchanged by the operation.

143.b/3 The root nodes do not contain element values, so they cannot be swapped.

143.c/3 **To be honest:** The implementation is not required to actually copy the elements if it can do the swap some other way. But it is allowed to copy the elements if needed.

```
{AI12-0112-1} function Find (Container : Tree;                                    144/5
                             Item      : Element_Type)
   return Cursor
   with Post => (if Find'Result /= No_Element
                 then Has_Element (Container, Find'Result));
```

{*AI05-0136-1*} {*AI05-0262-1*} Find searches the elements of Container for an element equal to Item (using the generic formal equality operator). The search starts at the root node. The search traverses the tree in a depth-first order. If no equal element is found, then Find returns No_Element. Otherwise, it returns a cursor designating the first equal element encountered. 145/3

```
function Find_In_Subtree (Position : Cursor;                                      146/5
                          Item     : Element_Type)
   return Cursor
   with Pre  => Position /= No_Element or else raise Constraint_Error,
        Post => (if Find_In_Subtree'Result = No_Element
                 then Has_Element (Find_In_Subtree'Result)),
        Global => in all;
```

{*AI05-0136-1*} {*AI05-0248-1*} {*AI05-0262-1*} {*AI12-0112-1*} Find_In_Subtree searches the subtree rooted by Position for an element equal to Item (using the generic formal equality operator). The search starts at the element designated by Position. The search traverses the subtree in a depth-first order. If no equal element is found, then Find returns No_Element. Otherwise, it returns a cursor designating the first equal element encountered. 147/5

Ramification: Find_In_Subtree does not check any siblings of the element designated by Position. The root node does not contain an element, and therefore it can never be returned, but it can be explicitly passed to Position. 147.a/3

```
{AI12-0112-1} function Find_In_Subtree (Container : Tree;                         147.1/5
                          Position  : Cursor;
                          Item      : Element_Type)
   return Cursor
   with Pre  => (Position /= No_Element
                    or else raise Constraint_Error) and then
                (Meaningful_For (Container, Position)
                    or else raise Program_Error),
        Post => (if Find_In_Subtree'Result = No_Element
                 then Has_Element (Container, Find_In_Subtree'Result));
```

{*AI12-0112-1*} Find_In_Subtree searches the subtree of Container rooted by Position for an element equal to Item (using the generic formal equality operator). The search starts at the element designated by Position. The search traverses the subtree in a depth-first order. If no equal element is found, then Find returns No_Element. Otherwise, it returns a cursor designating the first equal element encountered. 147.2/5

```
function Ancestor_Find (Position : Cursor;                                        148/5
                        Item     : Element_Type)
   return Cursor
   with Pre  => Position /= No_Element or else raise Constraint_Error,
        Post => (if Ancestor_Find'Result = No_Element
                 then Has_Element (Container, Ancestor_Find'Result)),
        Global => in all;
```

{*AI05-0136-1*} {*AI05-0248-1*} {*AI12-0112-1*} Ancestor_Find searches for an element equal to Item (using the generic formal equality operator). The search starts at the node designated by Position, and checks each ancestor proceeding toward the root of the subtree. If no equal element is found, then Ancestor_Find returns No_Element. Otherwise, it returns a cursor designating the first equal element encountered. 149/5

Ramification: {*AI05-0248-1*} No_Element is returned if Position is the root node. 149.a/3

149.1/5
```
{AI12-0112-1} function Ancestor_Find (Container : Tree;
                                      Position  : Cursor;
                                      Item      : Element_Type)
   return Cursor
   with Pre  => (Position /= No_Element
                    or else raise Constraint_Error) and then
                (Meaningful_For (Container, Position)
                    or else raise Program_Error),
        Post => (if Ancestor_Find'Result = No_Element
                 then Has_Element (Container, Ancestor_Find'Result));
```

149.2/5 {*AI12-0112-1*} Ancestor_Find searches for an element equal to Item (using the generic formal equality operator). The search starts at the node designated by Position in Container, and checks each ancestor proceeding toward the root of the subtree. If no equal element is found, then Ancestor_Find returns No_Element. Otherwise, it returns a cursor designating the first equal element encountered.

150/3
```
function Contains (Container : Tree;
                   Item      : Element_Type) return Boolean;
```

151/3 {*AI05-0136-1*} Equivalent to Find (Container, Item) /= No_Element.

152/5
```
{AI12-0112-1} procedure Iterate
  (Container : in Tree;
   Process   : not null access procedure (Position : in Cursor))
   with Allows_Exit;
```

153/4 {*AI05-0136-1*} {*AI05-0265-1*} {*AI12-0069-1*} Iterate calls Process.**all** with a cursor that designates each element in Container, starting from the root node and proceeding in a depth-first order. Tampering with the cursors of Container is prohibited during the execution of a call on Process.**all**. Any exception raised by Process.**all** is propagated.

153.a/3 **Ramification:** Process is not called with the root node, which does not have an associated element.

153.b/3 **Implementation Note:** The purpose of the tamper with cursors check is to prevent erroneous execution from the Position parameter of Process.**all** becoming invalid. This check takes place when the operations that tamper with the cursors of the container are called. The check cannot be made later (say in the body of Iterate), because that could cause the Position cursor to be invalid and potentially cause execution to become erroneous — defeating the purpose of the check.

153.c/3 See Iterate for vectors (A.18.2) for a suggested implementation of the check.

154/5
```
procedure Iterate_Subtree
  (Position  : in Cursor;
   Process   : not null access procedure (Position : in Cursor))
   with Allows_Exit,
        Pre  => Position /= No_Element or else raise Constraint_Error,
        Global => in all;
```

155/5 {*AI05-0136-1*} {*AI05-0265-1*} {*AI12-0069-1*} {*AI12-0112-1*} Iterate_Subtree calls Process.**all** with a cursor that designates each element in the subtree rooted by the node designated by Position, starting from the node designated by Position and proceeding in a depth-first order. Tampering with the cursors of the tree that contains the element designated by Position is prohibited during the execution of a call on Process.**all**. Any exception raised by Process.**all** is propagated.

155.a/3 **Ramification:** Position can be passed a cursor designating the root node; in that case, Process is not called with the root node, which does not have an associated element.

```
procedure Iterate_Subtree                                                    155.1/5
  (Container : in Tree;
   Position  : in Cursor;
   Process   : not null access procedure (Position : in Cursor))
   with Allows_Exit,
        Pre  => (Position /= No_Element
                   or else raise Constraint_Error) and then
                (Meaningful_For (Container, Position)
                   or else raise Program_Error);
```

{*AI12-0112-1*} Iterate_Subtree calls Process.**all** with a cursor that designates each element in the subtree rooted by the node designated by Position in Container, starting from the node designated by Position and proceeding in a depth-first order. Tampering with the cursors of the tree that contains the element designated by Position is prohibited during the execution of a call on Process.**all**. Any exception raised by Process.**all** is propagated. 155.2/5

```
{AI12-0112-1} function Iterate (Container : in Tree)                         156/5
   return Tree_Iterator_Interfaces.Parallel_Iterator'Class
   with Post => Tampering_With_Cursors_Prohibited (Container);
```

{*AI05-0212-1*} {*AI05-0265-1*} {*AI05-0269-1*} {*AI12-0069-1*} {*AI12-0266-1*} Iterate returns an iterator object (see 5.5.1) that will generate a value for a loop parameter (see 5.5.2) designating each element in Container, starting from the root node and proceeding in a depth-first order when used as a forward iterator, and processing all nodes concurrently when used as a parallel iterator. Tampering with the cursors of Container is prohibited while the iterator object exists (in particular, in the sequence_of_statements of the loop_statement whose iterator_specification denotes this object). The iterator object needs finalization. 157/5

Discussion: Exits are allowed from the loops created using the iterator objects. In particular, to stop the iteration at a particular cursor, just add 157.a/3

```
exit when Cur = Stop;                                                        157.b/3
```

in the body of the loop (assuming that `Cur` is the loop parameter and `Stop` is the cursor that you want to stop at). 157.c/3

```
function Iterate_Subtree (Position : in Cursor)                              158/5
   return Tree_Iterator_Interfaces.Parallel_Iterator'Class
   with Pre    => Position /= No_Element or else raise Constraint_Error,
        Global => in all;
```

{*AI05-0212-1*} {*AI05-0265-1*} {*AI05-0269-1*} {*AI12-0069-1*} {*AI12-0112-1*} {*AI12-0266-1*} Iterate_Subtree returns an iterator object (see 5.5.1) that will generate a value for a loop parameter (see 5.5.2) designating each element in the subtree rooted by the node designated by Position, starting from the node designated by Position and proceeding in a depth-first order when used as a forward iterator, and processing all nodes in the subtree concurrently when used as a parallel iterator. Tampering with the cursors of the container that contains the node designated by Position is prohibited while the iterator object exists (in particular, in the sequence_of_statements of the loop_statement whose iterator_specification denotes this object). The iterator object needs finalization. 159/5

```
function Iterate_Subtree (Container : in Tree; Position : in Cursor)         159.1/5
   return Tree_Iterator_Interfaces.Parallel_Iterator'Class
   with Pre  => (Position /= No_Element
                   or else raise Constraint_Error) and then
                (Meaningful_For (Container, Position)
                   or else raise Program_Error),
        Post => Tampering_With_Cursors_Prohibited (Container);
```

{*AI12-0112-1*} Iterate_Subtree returns an iterator object (see 5.5.1) that will generate a value for a loop parameter (see 5.5.2) designating each element in the subtree rooted by the node designated by Position in Container, starting from the node designated by Position and proceeding in a depth-first order when used as a forward iterator, and processing all nodes in the 159.2/5

subtree concurrently when used as a parallel iterator. Tampering with the cursors of the container that contains the node designated by Position is prohibited while the iterator object exists (in particular, in the sequence_of_statements of the loop_statement whose iterator_specification denotes this object). The iterator object needs finalization.

160/5
```
{AI12-0112-1} function Child_Count (Parent : Cursor) return Count_Type
   with Post => (if Parent = No_Element then Child_Count'Result = 0),
        Nonblocking, Global => in all, Use_Formal => null;
```

161/3 {*AI05-0136-1*} Child_Count returns the number of child nodes of the node designated by Parent.

161.1/5
```
function Child_Count (Container : Tree; Parent : Cursor)
   return Count_Type
   with Pre  => Meaningful_For (Container, Parent)
                   or else raise Program_Error,
        Post => (if Parent = No_Element then Child_Count'Result = 0),
        Nonblocking, Global => null, Use_Formal => null;
```

161.2/5 {*AI12-0112-1*} Child_Count returns the number of child nodes of the node designated by Parent in Container.

162/5
```
function Child_Depth (Parent, Child : Cursor) return Count_Type
   with Pre  => (Parent /= No_Element and then Child /= No_Element)
                   or else raise Constraint_Error,
     Nonblocking, Global => in all, Use_Formal => null;
```

163/5 {*AI05-0136-1*} {*AI05-0262-1*} {*AI12-0112-1*} Child_Depth returns the number of ancestor nodes of Child (including Child itself), up to but not including Parent; Program_Error is propagated if Parent is not an ancestor of Child.

163.a/3 **Ramification:** Program_Error is propagated if Parent and Child are nodes in different containers.

163.b/3 Child_Depth (Root (Some_Tree), Child) + 1 = Depth (Child) as the root is not counted.

163.1/5
```
function Child_Depth (Container : Tree; Parent, Child : Cursor)
   return Count_Type
   with Pre  => ((Parent /= No_Element and then Child /= No_Element)
                   or else raise Constraint_Error) and then
                (Meaningful_For (Container, Parent)
                   or else raise Program_Error) and then
                (Meaningful_For (Container, Child)
                   or else raise Program_Error),
          Nonblocking, Global => null, Use_Formal => null;
```

163.2/5 {*AI12-0112-1*} Child_Depth returns the number of ancestor nodes of Child within Container (including Child itself), up to but not including Parent; Program_Error is propagated if Parent is not an ancestor of Child.

```
procedure Insert_Child (Container : in out Tree;                                    164/5
                        Parent    : in     Cursor;
                        Before    : in     Cursor;
                        New_Item  : in     Element_Type;
                        Count     : in     Count_Type := 1)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                   or else raise Program_Error) and then
                 (Parent /= No_Element
                   or else raise Constraint_Error) and then
                 (Meaningful_For (Container, Parent)
                   or else raise Program_Error) and then
                 (Meaningful_For (Container, Before)
                   or else raise Program_Error) and then
                 (Before = No_Element or else
                  Container.Parent (Before) = Parent
                   or else raise Constraint_Error),
        Post => Node_Count (Container) =
                Node_Count (Container)'Old + Count;
```

{*AI05-0136-1*} {*AI05-0248-1*} {*AI05-0262-1*} {*AI12-0112-1*} Insert_Child allocates Count nodes containing copies of New_Item and inserts them as children of Parent. If Parent already has child nodes, then the new nodes are inserted prior to the node designated by Before, or, if Before equals No_Element, the new nodes are inserted after the last existing child node of Parent. Any exception raised during allocation of internal storage is propagated, and Container is not modified. 165/5

```
procedure Insert_Child (Container : in out Tree;                                    166/5
                        Parent    : in     Cursor;
                        Before    : in     Cursor;
                        New_Item  : in     Element_Type;
                        Position  :    out Cursor;
                        Count     : in     Count_Type := 1)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                   or else raise Program_Error) and then
                 (Parent /= No_Element
                   or else raise Constraint_Error) and then
                 (Meaningful_For (Container, Parent)
                   or else raise Program_Error) and then
                 (Meaningful_For (Container, Before)
                   or else raise Program_Error) and then
                 (Before = No_Element or else
                  Container.Parent (Before) = Parent
                   or else raise Constraint_Error),
        Post => (Node_Count (Container) =
                 Node_Count (Container)'Old + Count) and then
                 Has_Element (Container, Position);
```

{*AI05-0136-1*} {*AI05-0248-1*} {*AI05-0257-1*} {*AI05-0262-1*} {*AI12-0112-1*} Insert_Child allocates Count nodes containing copies of New_Item and inserts them as children of Parent. If Parent already has child nodes, then the new nodes are inserted prior to the node designated by Before, or, if Before equals No_Element, the new nodes are inserted after the last existing child node of Parent. Position designates the first newly-inserted node, or if Count equals 0, then Position is assigned the value of Before. Any exception raised during allocation of internal storage is propagated, and Container is not modified. 167/5

168/5
```
procedure Insert_Child (Container : in out Tree;
                        Parent    : in     Cursor;
                        Before    : in     Cursor;
                        Position  :    out Cursor;
                        Count     : in     Count_Type := 1)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                   or else raise Program_Error) and then
                (Parent /= No_Element
                   or else raise Constraint_Error) and then
                (Meaningful_For (Container, Parent)
                   or else raise Program_Error) and then
                (Meaningful_For (Container, Before)
                   or else raise Program_Error) and then
                (Before = No_Element or else
                 Container.Parent (Before) = Parent
                   or else raise Constraint_Error),
        Post => (Node_Count (Container) =
                 Node_Count (Container)'Old + Count) and then
                 Has_Element (Container, Position);
```

169/5 {*AI05-0136-1*} {*AI05-0257-1*} {*AI05-0262-1*} {*AI05-0264-1*} {*AI12-0112-1*} Insert_Child allocates Count nodes, the elements contained in the new nodes are initialized by default (see 3.3.1), and the new nodes are inserted as children of Parent. If Parent already has child nodes, then the new nodes are inserted prior to the node designated by Before, or, if Before equals No_Element, the new nodes are inserted after the last existing child node of Parent. Position designates the first newly-inserted node, or if Count equals 0, then Position is assigned the value of Before. Any exception raised during allocation of internal storage is propagated, and Container is not modified.

170/5
```
{AI12-0112-1} procedure Prepend_Child (Container : in out Tree;
                        Parent    : in     Cursor;
                        New_Item  : in     Element_Type;
                        Count     : in     Count_Type := 1)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                   or else raise Program_Error) and then
                (Parent /= No_Element
                   or else raise Constraint_Error) and then
                (Meaningful_For (Container, Parent)
                   or else raise Program_Error),
        Post => Node_Count (Container) =
                Node_Count (Container)'Old + Count;
```

171/3 {*AI05-0136-1*} Equivalent to Insert_Child (Container, Parent, First_Child (Container, Parent), New_Item, Count).

172/5
```
{AI12-0112-1} procedure Append_Child (Container : in out Tree;
                        Parent    : in     Cursor;
                        New_Item  : in     Element_Type;
                        Count     : in     Count_Type := 1)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                   or else raise Program_Error) and then
                (Parent /= No_Element
                   or else raise Constraint_Error) and then
                (Meaningful_For (Container, Parent)
                   or else raise Program_Error),
        Post => Node_Count (Container) =
                Node_Count (Container)'Old + Count;
```

173/3 {*AI05-0136-1*} {*AI05-0269-1*} Equivalent to Insert_Child (Container, Parent, No_Element, New_Item, Count).

```
procedure Delete_Children (Container : in out Tree;                                    174/5
                           Parent    : in     Cursor)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error) and then
                 (Parent /= No_Element
                    or else raise Constraint_Error) and then
                 (Meaningful_For (Container, Parent)
                    or else raise Program_Error),
        Post => (Node_Count (Container) = Node_Count (Container)'Old -
                    Child_Count (Container, Parent)'Old) and then
                  Child_Count (Container, Parent) = 0;
```

{*AI05-0136-1*} {*AI12-0112-1*} Delete_Children removes (from Container) all of the descendants of Parent other than Parent itself. 175/5

Discussion: This routine deletes all of the child subtrees of Parent at once. Use Delete_Subtree to delete an individual subtree. 175.a/3

```
procedure Copy_Subtree (Target   : in out Tree;                                        176/5
                        Parent   : in     Cursor;
                        Before   : in     Cursor;
                        Source   : in     Cursor)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Target)
                    or else raise Program_Error) and then
                 (Parent /= No_Element
                    or else raise Constraint_Error) and then
                 (Meaningful_For (Target, Parent)
                    or else raise Program_Error) and then
                 (Meaningful_For (Target, Before)
                    or else raise Program_Error) and then
                 (Before = No_Element or else
                    Target.Parent (Before) = Parent
                    or else raise Constraint_Error) and then
                 (not Is_Root (Source)
                    or else raise Constraint_Error),
        Post => Node_Count (Target) =
                Node_Count (Target)'Old + Subtree_Node_Count (Source),
        Global => in all;
```

{*AI05-0136-1*} {*AI05-0248-1*} {*AI05-0262-1*} {*AI12-0112-1*} If Source is equal to No_Element, then the operation has no effect. Otherwise, the subtree rooted by Source (which can be from any tree; it does not have to be a subtree of Target) is copied (new nodes are allocated to create a new subtree with the same structure as the Source subtree, with each element initialized from the corresponding element of the Source subtree) and inserted into Target as a child of Parent. If Parent already has child nodes, then the new nodes are inserted prior to the node designated by Before, or, if Before equals No_Element, the new nodes are inserted after the last existing child node of Parent. The parent of the newly created subtree is set to Parent, and the overall count of Target is incremented by Subtree_Node_Count (Source). Any exception raised during allocation of internal storage is propagated, and Container is not modified. 177/5

Discussion: {*AI12-0112-1*} We only need one routine here, as the source object is not modified, so we can use the same routine for both copying within and between containers. However, that requires a contract that allows reading of any container of the correct type, so we provide two other routines wuth more restrictive contracts. 177.a/5

Ramification: We do not allow copying a subtree that includes a root node, as that would require inserting a node with no value in the middle of the target tree. To copy an entire tree to another tree object, use Copy. 177.b/3

177.1/5
```
procedure Copy_Local_Subtree (Target   : in out Tree;
                              Parent   : in     Cursor;
                              Before   : in     Cursor;
                              Source   : in     Cursor)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Target)
                    or else raise Program_Error) and then
                 (Parent /= No_Element
                    or else raise Constraint_Error) and then
                 (Meaningful_For (Target, Parent)
                    or else raise Program_Error) and then
                 (Meaningful_For (Target, Before)
                    or else raise Program_Error) and then
                 (Before = No_Element or else
                    Target.Parent (Before) = Parent
                    or else raise Constraint_Error) and then
                 (Meaningful_For (Target, Source)
                    or else raise Program_Error) and then
                 (not Is_Root (Source)
                    or else raise Constraint_Error),
        Post => Node_Count (Target) = Node_Count (Target)'Old +
                  Subtree_Node_Count (Target, Source);
```

177.2/5 {*AI12-0112-1*} If Source is equal to No_Element, then the operation has no effect. Otherwise, the subtree rooted by Source in Target is copied (new nodes are allocated to create a new subtree with the same structure as the Source subtree, with each element initialized from the corresponding element of the Source subtree) and inserted into Target as a child of Parent. If Parent already has child nodes, then the new nodes are inserted prior to the node designated by Before, or, if Before equals No_Element, the new nodes are inserted after the last existing child node of Parent. The parent of the newly created subtree is set to Parent. Any exception raised during allocation of internal storage is propagated, and Container is not modified.

177.3/5
```
procedure Copy_Subtree (Target   : in out Tree;
                        Parent   : in     Cursor;
                        Before   : in     Cursor;
                        Source   : in     Tree;
                        Subtree  : in     Cursor)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Target)
                    or else raise Program_Error) and then
                 (Parent /= No_Element
                    or else raise Constraint_Error) and then
                 (Meaningful_For (Target, Parent)
                    or else raise Program_Error) and then
                 (Meaningful_For (Target, Before)
                    or else raise Program_Error) and then
                 (Before = No_Element or else
                    Target.Parent (Before) = Parent
                    or else raise Constraint_Error) and then
                 (Meaningful_For (Source, Subtree)
                    or else raise Program_Error) and then
                 (not Is_Root (Source, Subtree)
                    or else raise Constraint_Error),
        Post => Node_Count (Target) = Node_Count (Target)'Old +
                  Subtree_Node_Count (Source, Subtree);
```

177.4/5 {*AI12-0112-1*} If Subtree is equal to No_Element, then the operation has no effect. Otherwise, the subtree rooted by Subtree in Source is copied (new nodes are allocated to create a new subtree with the same structure as the Subtree, with each element initialized from the corresponding element of the Subtree) and inserted into Target as a child of Parent. If Parent already has child nodes, then the new nodes are inserted prior to the node designated by Before, or, if Before equals No_Element, the new nodes are inserted after the last existing child node of Parent. The parent of the newly created subtree is set to Parent. Any exception raised during allocation of internal storage is propagated, and Container is not modified.

```
procedure Splice_Subtree (Target   : in out Tree;                                  178/5
                          Parent   : in     Cursor;
                          Before   : in     Cursor;
                          Source   : in out Tree;
                          Position : in out Cursor)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Target)
                    or else raise Program_Error) and then
                 (not Tampering_With_Cursors_Prohibited (Source)
                    or else raise Program_Error) and then
                 (Parent /= No_Element
                    or else raise Constraint_Error) and then
                 (Meaningful_For (Target, Parent)
                    or else raise Program_Error) and then
                 (Meaningful_For (Target, Before)
                    or else raise Program_Error) and then
                 (Before = No_Element or else
                  Target.Parent (Before) /= Parent
                    or else raise Constraint_Error) and then
                 (Position /= No_Element
                    or else raise Constraint_Error) and then
                 (Has_Element (Source, Position)
                    or else raise Program_Error) and then
                 (Target'Has_Same_Storage (Source) or else
                  Position = Before or else
                  Is_Ancestor_Of (Target, Position, Parent)
                    or else raise Constraint_Error),
        Post => (declare
                    Org_Sub_Count renames
                        Subtree_Node_Count (Source, Position)'Old;
                    Org_Target_Count renames Node_Count (Target)'Old;
                 begin
                   (if not Target'Has_Same_Storage (Source) then
                       Node_Count (Target) = Org_Target_Count +
                          Org_Sub_Count and then
                       Node_Count (Source) = Node_Count (Source)'Old -
                          Org_Sub_Count and then
                       Has_Element (Target, Position)
                    else
                       Target.Parent (Position) = Parent and then
                       Node_Count (Target) = Org_Target_Count));
```

{*AI05-0136-1*} {*AI05-0248-1*} {*AI05-0262-1*} {*AI05-0269-1*} {*AI12-0112-1*} If Source denotes the same object as Target, then: if Position equals Before there is no effect; otherwise, the subtree rooted by the element designated by Position is moved to be a child of Parent. If Parent already has child nodes, then the moved nodes are inserted prior to the node designated by Before, or, if Before equals No_Element, the moved nodes are inserted after the last existing child node of Parent. In each of these cases, Position and the count of Target are unchanged, and the parent of the element designated by Position is set to Parent. 179/5

Reason: We can't allow moving the subtree of Position to a proper descendant node of the subtree, as the descendant node will be part of the subtree being moved. The result would be a circularly linked tree, or one with inaccessible nodes. Thus we have to check Position against Parent, even though such a check is *O*(Depth(Source)). 179.a/3

{*AI05-0136-1*} {*AI05-0248-1*} Otherwise (if Source does not denote the same object as Target), the subtree designated by Position is removed from Source and moved to Target. The subtree is inserted as a child of Parent. If Parent already has child nodes, then the moved nodes are inserted prior to the node designated by Before, or, if Before equals No_Element, the moved nodes are inserted after the last existing child node of Parent. In each of these cases, the count of Target is incremented by Subtree_Node_Count (Position), and the count of Source is decremented by Subtree_Node_Count (Position), Position is updated to represent an element in Target. 180/3

Ramification: If Source is the same as Target, and Position = Before, or Next_Sibling(Position) = Before, Splice_Subtree has no effect, as the subtree does not have to move to meet the postcondition. 180.a/3

180.b/3 We do not allow splicing a subtree that includes a root node, as that would require inserting a node with no value in the middle of the target tree. Splice the children of the root node instead.

180.c/3 For this reason there is no operation to splice an entire tree, as that would necessarily involve splicing a root node.

181/5
```
procedure Splice_Subtree (Container: in out Tree;
                          Parent   : in     Cursor;
                          Before   : in     Cursor;
                          Position : in     Cursor)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error) and then
                 (Parent /= No_Element
                    or else raise Constraint_Error) and then
                 (Meaningful_For (Container, Parent)
                    or else raise Program_Error) and then
                 (Meaningful_For (Container, Before)
                    or else raise Program_Error) and then
                 (Before = No_Element or else
                  Container.Parent (Before) /= Parent
                    or else raise Constraint_Error) and then
                 (Position /= No_Element
                    or else raise Constraint_Error) and then
                 (Has_Element (Container, Position)
                    or else raise Program_Error) and then
                 (Position = Before or else
                  Is_Ancestor_Of (Container, Position, Parent)
                    or else raise Constraint_Error),
        Post => (Node_Count (Container) =
                    Node_Count (Container)'Old and then
                  Container.Parent (Position) = Parent);
```

182/5 {*AI05-0136-1*} {*AI05-0248-1*} {*AI05-0262-1*} {*AI05-0269-1*} {*AI12-0112-1*} If Position equals Before, there is no effect. Otherwise, the subtree rooted by the element designated by Position is moved to be a child of Parent. If Parent already has child nodes, then the moved nodes are inserted prior to the node designated by Before, or, if Before equals No_Element, the moved nodes are inserted after the last existing child node of Parent. The parent of the element designated by Position is set to Parent.

182.a/3 **Reason:** We can't allow moving the subtree of Position to a proper descendant node of the subtree, as the descendant node will be part of the subtree being moved.

```
procedure Splice_Children (Target          : in out Tree;                    183/5
                           Target_Parent   : in     Cursor;
                           Before          : in     Cursor;
                           Source          : in out Tree;
                           Source_Parent   : in     Cursor)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Target)
                    or else raise Program_Error) and then
                 (not Tampering_With_Cursors_Prohibited (Source)
                    or else raise Program_Error) and then
                 (Target_Parent /= No_Element
                    or else raise Constraint_Error) and then
                 (Meaningful_For (Target, Target_Parent)
                    or else raise Program_Error) and then
                 (Meaningful_For (Target, Before)
                    or else raise Program_Error) and then
                 (Source_Parent /= No_Element
                    or else raise Constraint_Error) and then
                 (Meaningful_For (Source, Source_Parent)
                    or else raise Program_Error) and then
                 (Before = No_Element or else
                  Parent (Target, Before) /= Target_Parent
                    or else raise Constraint_Error) and then
                 (Target'Has_Same_Storage (Source) or else
                  Target_Parent = Source_Parent or else
                  Is_Ancestor_Of (Target, Source_Parent, Target_Parent)
                    or else raise Constraint_Error),
        Post => (declare
                    Org_Child_Count renames
                       Child_Count (Source, Source_Parent)'Old;
                    Org_Target_Count renames Node_Count (Target)'Old;
                 begin
                   (if not Target'Has_Same_Storage (Source) then
                       Node_Count (Target) = Org_Target_Count +
                          Org_Child_Count and then
                       Node_Count (Source) = Node_Count (Source)'Old -
                          Org_Child_Count
                    else
                       Node_Count (Target) = Org_Target_Count));
```

This paragraph was deleted.{*AI05-0136-1*} {*AI05-0262-1*} {*AI12-0112-1*} 184/5

If Source denotes the same object as Target, then: 185/3

- if Target_Parent equals Source_Parent there is no effect; else 186/3
- *This paragraph was deleted.*{*AI05-0136-1*} {*AI05-0269-1*} {*AI12-0112-1*} 187/5
- {*AI05-0136-1*} {*AI05-0248-1*} {*AI05-0269-1*} the child elements (and the further descendants) of Source_Parent are moved to be child elements of Target_Parent. If Target_Parent already has child elements, then the moved elements are inserted prior to the node designated by Before, or, if Before equals No_Element, the moved elements are inserted after the last existing child node of Target_Parent. The parent of each moved child element is set to Target_Parent. 188/3

Reason: We can't allow moving the children of Source_Parent to a proper descendant node, as the descendant node will be part of one of the subtrees being moved. 188.a/3

{*AI05-0136-1*} {*AI05-0248-1*} {*AI05-0269-1*} Otherwise (if Source does not denote the same object as Target), the child elements (and the further descendants) of Source_Parent are removed from Source and moved to Target. The child elements are inserted as children of Target_Parent. If Target_Parent already has child elements, then the moved elements are inserted prior to the node designated by Before, or, if Before equals No_Element, the moved elements are inserted after the last existing child node of Target_Parent. In each of these cases, the overall count of Target is incremented by Subtree_Node_Count (Source_Parent)-1, and the overall count of Source is decremented by Subtree_Node_Count (Source_Parent)-1. 189/3

189.a/3 **Ramification:** The node designated by Source_Parent is not moved, thus we never need to update Source_Parent.

189.b/3 Move (Target, Source) could be written Splice_Children (Target, Target.Root, No_Element, Source, Source.Root);

190/5
```
procedure Splice_Children (Container       : in out Tree;
                           Target_Parent   : in     Cursor;
                           Before          : in     Cursor;
                           Source_Parent   : in     Cursor)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                   or else raise Program_Error) and then
                (Target_Parent /= No_Element
                   or else raise Constraint_Error) and then
                (Meaningful_For (Container, Target_Parent)
                   or else raise Program_Error) and then
                (Meaningful_For (Container, Before)
                   or else raise Program_Error) and then
                (Source_Parent /= No_Element
                   or else raise Constraint_Error) and then
                (Meaningful_For (Container, Source_Parent)
                   or else raise Program_Error) and then
                (Before = No_Element or else
                 Parent (Container, Before) /= Target_Parent
                   or else raise Constraint_Error) and then
                (Target_Parent = Source_Parent or else
                 Is_Ancestor_Of (Container, Source_Parent, Target_Parent)
                   or else raise Constraint_Error),
        Post => Node_Count (Container) = Node_Count (Container)'Old;
```

191/5 {*AI05-0136-1*} {*AI05-0248-1*} {*AI05-0262-1*} {*AI05-0264-1*} {*AI05-0269-1*} {*AI12-0112-1*} If Target_Parent equals Source_Parent there is no effect. Otherwise, the child elements (and the further descendants) of Source_Parent are moved to be child elements of Target_Parent. If Target_Parent already has child elements, then the moved elements are inserted prior to the node designated by Before, or, if Before equals No_Element, the moved elements are inserted after the last existing child node of Target_Parent. The parent of each moved child element is set to Target_Parent.

192/5
```
function Parent (Position : Cursor) return Cursor
   with Nonblocking, Global => in all, Use_Formal => null,
        Post => (if Position = No_Element or else
                    Is_Root (Position) then Parent'Result = No_Element);
```

193/5 {*AI05-0136-1*} {*AI12-0112-1*} Returns a cursor designating the parent node of the node designated by Position.

193.1/5
```
function Parent (Container : Tree;
                 Position  : Cursor) return Cursor
   with Nonblocking, Global => null, Use_Formal => null,
        Pre  => Meaningful_For (Container, Position)
                   or else raise Program_Error,
        Post => (if Position = No_Element or else
                   Is_Root (Container, Position)
                   then Parent'Result = No_Element
                  else Has_Element (Container, Parent'Result));
```

193.2/5 {*AI12-0112-1*} Returns a cursor designating the parent node of the node designated by Position in Container.

194/5
```
function First_Child (Parent : Cursor) return Cursor
   with Nonblocking, Global => in all, Use_Formal => null,
        Pre  => Parent /= No_Element or else raise Constraint_Error;
```

195/5 {*AI05-0136-1*} {*AI12-0112-1*} First_Child returns a cursor designating the first child node of the node designated by Parent; if there is no such node, No_Element is returned.

```
function First_Child (Container : Tree;                                    195.1/5
                      Parent    : Cursor) return Cursor
   with Nonblocking, Global => null, Use_Formal => null,
        Pre  => (Parent /= No_Element
                   or else raise Constraint_Error) and then
                 (Meaningful_For (Container, Parent)
                   or else raise Program_Error),
        Post => First_Child'Result = No_Element or else
                Has_Element (Container, First_Child'Result);
```

{*AI12-0112-1*} First_Child returns a cursor designating the first child node of the node designated by Parent in Container; if there is no such node, No_Element is returned. 195.2/5

```
{AI12-0112-1} function First_Child_Element (Parent : Cursor) return Element_Type   196/5
   with Nonblocking, Global => in all, Use_Formal => Element_Type,
        Pre  => (Parent /= No_Element and then
                 Last_Child (Parent) /= No_Element)
                     or else raise Constraint_Error;
```

{*AI05-0136-1*} Equivalent to Element (First_Child (Parent)). 197/3

```
function First_Child_Element (Container : Tree;                            197.1/5
                              Parent    : Cursor) return Element_Type
   with Nonblocking, Global => null, Use_Formal => Element_Type,
        Pre  => (Parent /= No_Element
                   or else raise Constraint_Error) and then
                 (Meaningful_For (Container, Parent)
                   or else raise Program_Error) and then
                 (First_Child (Container, Parent) /= No_Element
                   or else raise Constraint_Error);
```

{*AI12-0112-1*} Equivalent to Element (Container, First_Child (Container, Parent)). 197.2/5

```
function Last_Child (Parent : Cursor) return Cursor                        198/5
   with Nonblocking, Global => in all, Use_Formal => null,
        Pre  => Parent /= No_Element or else raise Constraint_Error;
```

{*AI05-0136-1*} {*AI12-0112-1*} Last_Child returns a cursor designating the last child node of the node designated by Parent; if there is no such node, No_Element is returned. 199/5

```
function Last_Child (Container : Tree;                                     199.1/5
                     Parent    : Cursor) return Cursor
   with Nonblocking, Global => null, Use_Formal => null,
        Pre  => (Parent /= No_Element
                   or else raise Constraint_Error) and then
                 (Meaningful_For (Container, Parent)
                   or else raise Program_Error),
        Post => Last_Child'Result = No_Element or else
                Has_Element (Container, Last_Child'Result);
```

{*AI12-0112-1*} Last_Child returns a cursor designating the last child node of the node designated by Parent in Container; if there is no such node, No_Element is returned. 199.2/5

```
{AI12-0112-1} function Last_Child_Element (Parent : Cursor) return Element_Type   200/5
   with Nonblocking, Global => in all, Use_Formal => Element_Type,
        Pre  => (Parent /= No_Element and then
                 Last_Child (Parent) /= No_Element)
                     or else raise Constraint_Error;
```

{*AI05-0136-1*} Equivalent to Element (Last_Child (Parent)). 201/3

201.1/5
```
function Last_Child_Element (Container : Tree;
                             Parent    : Cursor) return Element_Type
   with Nonblocking, Global => null, Use_Formal => Element_Type,
        Pre  => (Parent /= No_Element
                   or else raise Constraint_Error) and then
                 (Meaningful_For (Container, Parent)
                   or else raise Program_Error) and then
                 (Last_Child (Container, Parent) /= No_Element
                   or else raise Constraint_Error);
```

201.2/5 {*AI12-0112-1*} Equivalent to Element (Container, Last_Child (Container, Parent)).

202/5
```
{AI12-0112-1} function Next_Sibling (Position : Cursor) return Cursor
   with Nonblocking, Global => in all, Use_Formal => null,
        Post => (if Position = No_Element
                    then Next_Sibling'Result = No_Element);
```

203/3 {*AI05-0136-1*} If Position equals No_Element or designates the last child node of its parent, then Next_Sibling returns the value No_Element. Otherwise, it returns a cursor that designates the successor (with the same parent) of the node designated by Position.

203.1/5
```
function Next_Sibling (Container : Tree;
                       Position  : Cursor) return Cursor
   with Nonblocking, Global => null, Use_Formal => null,
        Pre  => Meaningful_For (Container, Position)
                   or else raise Program_Error,
        Post => (if Next_Sibling'Result = No_Element then
                   Position = No_Element or else
                   Is_Root (Container, Position) or else
                   Last_Child (Container, Parent (Container, Position))
                      = Position
                 else Has_Element (Container, Next_Sibling'Result));
```

203.2/5 {*AI12-0112-1*} Next_Sibling returns a cursor that designates the successor (with the same parent) of the node designated by Position in Container.

204/5
```
{AI12-0112-1} function Previous_Sibling (Position : in out Cursor)
   with Nonblocking, Global => in all, Use_Formal => null,
        Post => (if Position = No_Element
                    then Previous_Sibling'Result = No_Element);
```

205/3 {*AI05-0136-1*} If Position equals No_Element or designates the first child node of its parent, then Previous_Sibling returns the value No_Element. Otherwise, it returns a cursor that designates the predecessor (with the same parent) of the node designated by Position.

205.1/5
```
function Previous_Sibling (Container : Tree;
                           Position  : Cursor) return Cursor
   with Nonblocking, Global => null, Use_Formal => null,
        Pre  => Meaningful_For (Container, Position)
                   or else raise Program_Error,
        Post => (if Previous_Sibling'Result = No_Element then
                   Position = No_Element or else
                   Is_Root (Container, Position) or else
                   First_Child (Container, Parent (Container, Position))
                      = Position
                 else Has_Element (Container, Previous_Sibling'Result));
```

205.2/5 {*AI12-0112-1*} Previous_Sibling returns a cursor that designates the predecessor (with the same parent) of the node designated by Position in Container.

206/5
```
{AI12-0112-1} procedure Next_Sibling (Position : in out Cursor)
   with Nonblocking, Global => in all, Use_Formal => null;
```

207/3 {*AI05-0136-1*} Equivalent to Position := Next_Sibling (Position);

```
procedure Next_Sibling (Container : in     Tree;                                    207.1/5
                        Position  : in out Cursor)
   with Nonblocking, Global => null, Use_Formal => null,
        Pre  => Meaningful_For (Container, Position)
                  or else raise Program_Error,
        Post => (if Position /= No_Element
                 then Has_Element (Container, Position));
```

{*AI12-0112-1*} Equivalent to Position := Next_Sibling (Container, Position); 207.2/5

```
{AI12-0112-1} procedure Previous_Sibling (Position : in out Cursor)                  208/5
   with Nonblocking, Global => in all, Use_Formal => null;
```

{*AI05-0136-1*} Equivalent to Position := Previous_Sibling (Position); 209/3

```
procedure Previous_Sibling (Container : in     Tree;                                209.1/5
                            Position  : in out Cursor)
   with Nonblocking, Global => null, Use_Formal => null,
        Pre  => Meaningful_For (Container, Position)
                  or else raise Program_Error,
        Post => (if Position /= No_Element
                 then Has_Element (Container, Position);
```

{*AI12-0112-1*} Equivalent to Position := Previous_Sibling (Container, Position); 209.2/5

```
{AI12-0112-1} procedure Iterate_Children                                             210/5
     (Parent  : in Cursor;
      Process : not null access procedure (Position : in Cursor))
   with Allows_Exit,
        Pre    => Parent /= No_Element or else raise Constraint_Error,
        Global => in all, Use_Formal => null;
```

This paragraph was deleted.{*AI05-0136-1*} {*AI05-0248-1*} {*AI12-0112-1*} 211/5

Iterate_Children calls Process.**all** with a cursor that designates each child node of Parent, starting with the first child node and moving the cursor as per the Next_Sibling function. 212/3

{*AI05-0265-1*} Tampering with the cursors of the tree containing Parent is prohibited during the execution of a call on Process.**all**. Any exception raised by Process.**all** is propagated. 213/3

```
{AI12-0112-1} procedure Iterate_Children                                             213.1/5
     (Container : in Tree;
      Parent    : in Cursor;
      Process   : not null access procedure (Position : in Cursor))
   with Allows_Exit,
        Pre  => (Parent /= No_Element
                   or else raise Constraint_Error) and then
                 (Meaningful_For (Container, Parent)
                   or else raise Program_Error);
```

{*AI12-0112-1*} Iterate_Children calls Process.**all** with a cursor that designates each child node of Container and Parent, starting with the first child node and moving the cursor as per the Next_Sibling function. 213.2/5

{*AI12-0112-1*} Tampering with the cursors of the tree containing Parent is prohibited during the execution of a call on Process.**all**. Any exception raised by Process.**all** is propagated. 213.3/5

```
{AI12-0112-1} procedure Reverse_Iterate_Children                                     214/5
     (Parent  : in Cursor;
      Process : not null access procedure (Position : in Cursor))
   with Allows_Exit,
        Pre  => Parent /= No_Element or else raise Constraint_Error,
        Global => in all, Use_Formal => null;
```

This paragraph was deleted.{*AI05-0136-1*} {*AI05-0248-1*} {*AI12-0112-1*} 215/5

216/3 Reverse_Iterate_Children calls Process.**all** with a cursor that designates each child node of Parent, starting with the last child node and moving the cursor as per the Previous_Sibling function.

217/3 {*AI05-0265-1*} Tampering with the cursors of the tree containing Parent is prohibited during the execution of a call on Process.**all**. Any exception raised by Process.**all** is propagated.

217.1/5
```
{AI12-0112-1} procedure Reverse_Iterate_Children
     (Container : in Tree;
      Parent    : in Cursor;
      Process   : not null access procedure (Position : in Cursor))
   with Allows_Exit,
        Pre  => (Parent /= No_Element
                   or else raise Constraint_Error) and then
                (Meaningful_For (Container, Parent)
                   or else raise Program_Error);
```

217.2/5 {*AI12-0112-1*} Reverse_Iterate_Children calls Process.**all** with a cursor that designates each child node of Container and Parent, starting with the last child node and moving the cursor as per the Previous_Sibling function.

217.3/5 {*AI12-0112-1*} Tampering with the cursors of the tree containing Parent is prohibited during the execution of a call on Process.**all**. Any exception raised by Process.**all** is propagated.

218/5
```
function Iterate_Children (Container : in Tree; Parent : in Cursor)
   return Tree_Iterator_Interfaces.Parallel_Reversible_Iterator'Class
   with Pre  => (Parent /= No_Element
                   or else raise Constraint_Error) and then
                (Meaningful_For (Container, Parent)
                   or else raise Program_Error),
        Post => Tampering_With_Cursors_Prohibited (Container);
```

219/5 {*AI05-0212-1*} {*AI05-0265-1*} {*AI12-0112-1*} {*AI12-0266-1*} Iterate_Children returns an iterator object (see 5.5.1) that will generate a value for a loop parameter (see 5.5.2) designating each child node of Parent. When used as a forward iterator, the nodes are designated starting with the first child node and moving the cursor as per the function Next_Sibling; when used as a reverse iterator, the nodes are designated starting with the last child node and moving the cursor as per the function Previous_Sibling; when used as a parallel iterator, processing all child nodes concurrently. Tampering with the cursors of Container is prohibited while the iterator object exists (in particular, in the sequence_of_statements of the loop_statement whose iterator_specification denotes this object). The iterator object needs finalization.

219.1/5 {*AI12-0111-1*} The nested package Multiway_Trees.Stable provides a type Stable.Tree that represents a *stable* tree, which is one that cannot grow and shrink. Such a tree can be created by calling the Copy function, or by establishing a *stabilized view* of an ordinary tree.

219.2/5 {*AI12-0111-1*} The subprograms of package Containers.Multiway_Trees that have a parameter or result of type tree are included in the nested package Stable with the same specification, except that the following are omitted:

219.3/5 Tampering_With_Cursors_Prohibited, Tampering_With_Elements_Prohibited, Assign, Move, Clear, Delete_Leaf, Insert_Child, Delete_Children, Delete_Subtree, Copy_Subtree, Copy_Local_Subtree, Splice_Subtree, and Splice_Children

219.a.1/5 **Ramification:** The names Tree and Cursor mean the types declared in the nested package in these subprogram specifications.

219.a.2/5 **Reason:** The omitted routines are those that tamper with cursors or elements (or test that state). The model is that it is impossible to tamper with cursors or elements of a stable view since no such operations are included. Thus tampering checks are not needed for a stable view, and we omit the operations associated with those checks.

{*AI12-0111-1*} The operations of this package are equivalent to those for ordinary trees, except that the calls to Tampering_With_Cursors_Prohibited and Tampering_With_Elements_Prohibited that occur in preconditions are replaced by False, and any that occur in postconditions are replaced by True. 219.4/5

{*AI12-0111-1*} {*AI12-0439-1*} If a stable tree is declared with the Base discriminant designating a pre-existing ordinary tree, the stable tree represents a stabilized view of the underlying ordinary tree, and any operation on the stable tree is reflected on the underlying ordinary tree. While a stabilized view exists, any operation that tampers with elements performed on the underlying tree is prohibited. The finalization of a stable tree that provides such a view removes this restriction on the underlying ordinary tree [(though some other restriction can exist due to other concurrent iterations or stabilized views)]. 219.5/5

{*AI12-0111-1*} {*AI12-0438-1*} If a stable tree is declared without specifying Base, the object is necessarily initialized. The initializing expression of the stable tree, [typically a call on Copy], determines the Node_Count of the tree. The Node_Count of a stable tree never changes after initialization. 219.6/5

Proof: {*AI12-0438-1*} Initialization is required as the type is indefinite, see 3.3.1. 219.a/5

Bounded (Run-Time) Errors

{*AI05-0136-1*} {*AI05-0248-1*} It is a bounded error for the actual function associated with a generic formal subprogram, when called as part of an operation of this package, to tamper with elements of any Tree parameter of the operation. Either Program_Error is raised, or the operation works as defined on the value of the Tree either prior to, or subsequent to, some or all of the modifications to the Tree. 220/3

{*AI05-0136-1*} It is a bounded error to call any subprogram declared in the visible part of Containers.Multiway_Trees when the associated container has been finalized. If the operation takes Container as an **in out** parameter, then it raises Constraint_Error or Program_Error. Otherwise, the operation either proceeds as it would for an empty container, or it raises Constraint_Error or Program_Error. 221/3

Erroneous Execution

{*AI05-0136-1*} A Cursor value is *invalid* if any of the following have occurred since it was created: 222/3

- The tree that contains the element it designates has been finalized; 223/3
- The tree that contains the element it designates has been used as the Source or Target of a call to Move; 224/3
- The tree that contains the element it designates has been used as the Target of a call to Assign or the target of an assignment_statement; 225/3
- The element it designates has been removed from the tree that previously contained the element. 226/3

Reason: We talk about which tree the element was removed from in order to handle splicing nodes from one tree to another. The node still exists, but any cursors that designate it in the original tree are now invalid. This bullet covers removals caused by calls to Clear, Delete_Leaf, Delete_Subtree, Delete_Children, Splice_Children, and Splice_Subtree. 226.a/3

The result of "=" or Has_Element is unspecified if it is called with an invalid cursor parameter. Execution is erroneous if any other subprogram declared in Containers.Multiway_Trees is called with an invalid cursor parameter. 227/3

Discussion: The list above is intended to be exhaustive. In other cases, a cursor value continues to designate its original element (or the root node). For instance, cursor values survive the insertion and deletion of other nodes. 227.a/3

While it is possible to check for these cases, in many cases the overhead necessary to make the check is substantial in time or space. Implementations are encouraged to check for as many of these cases as possible and raise Program_Error if detected. 227.b/3

228/3 {*AI05-0212-1*} Execution is erroneous if the tree associated with the result of a call to Reference or Constant_Reference is finalized before the result object returned by the call to Reference or Constant_Reference is finalized.

228.a/3 **Reason:** Each object of Reference_Type and Constant_Reference_Type probably contains some reference to the originating container. If that container is prematurely finalized (which is only possible via Unchecked_Deallocation, as accessibility checks prevent passing a container to Reference that will not live as long as the result), the finalization of the object of Reference_Type will try to access a nonexistent object. This is a normal case of a dangling pointer created by Unchecked_Deallocation; we have to explicitly mention it here as the pointer in question is not visible in the specification of the type. (This is the same reason we have to say this for invalid cursors.)

Implementation Requirements

229/3 {*AI05-0136-1*} No storage associated with a multiway tree object shall be lost upon assignment or scope exit.

230/3 {*AI05-0136-1*} {*AI05-0262-1*} The execution of an assignment_statement for a tree shall have the effect of copying the elements from the source tree object to the target tree object and changing the node count of the target object to that of the source object.

230.a/3 **Implementation Note:** {*AI05-0298-1*} An assignment of a Tree is a "deep" copy; that is the elements are copied as well the data structures. We say "effect of" in order to allow the implementation to avoid copying elements immediately if it wishes. For instance, an implementation that avoided copying until one of the containers is modified would be allowed. (Note that this implementation would require care, see A.18.2 for more.)

Implementation Advice

231/3 {*AI05-0136-1*} Containers.Multiway_Trees should be implemented similarly to a multiway tree. In particular, if *N* is the overall number of nodes for a particular tree, then the worst-case time complexity of Element, Parent, First_Child, Last_Child, Next_Sibling, Previous_Sibling, Insert_Child with Count=1, and Delete should be $O(\log N)$.

231.a/3 **Implementation Advice:** The worst-case time complexity of the Element, Parent, First_Child, Last_Child, Next_Sibling, Previous_Sibling, Insert_Child with Count=1, and Delete operations of Containers.Multiway_Trees should be $O(\log N)$.

231.b/3 **Reason:** We do not mean to overly constrain implementation strategies here. However, it is important for portability that the performance of large containers has roughly the same factors on different implementations. If a program is moved to an implementation that takes $O(N)$ time to access elements, that program could be unusable when the trees are large. We allow $O(\log N)$ access because the proportionality constant and caching effects are likely to be larger than the log factor, and we don't want to discourage innovative implementations.

232/3 {*AI05-0136-1*} Move should not copy elements, and should minimize copying of internal data structures.

232.a/3 **Implementation Advice:** Containers.Multiway_Trees.Move should not copy elements, and should minimize copying of internal data structures.

232.b/3 **Implementation Note:** Usually that can be accomplished simply by moving the pointer(s) to the internal data structures from the Source container to the Target container.

233/3 {*AI05-0136-1*} If an exception is propagated from a tree operation, no storage should be lost, nor any elements removed from a tree unless specified by the operation.

233.a/3 **Implementation Advice:** If an exception is propagated from a tree operation, no storage should be lost, nor any elements removed from a tree unless specified by the operation.

233.b/3 **Reason:** This is important so that programs can recover from errors. But we don't want to require heroic efforts, so we just require documentation of cases where this can't be accomplished.

Extensions to Ada 2005

233.c/3 {*AI05-0136-1*} {*AI05-0257-1*} {*AI05-0265-1*} {*AI05-0269-1*} The generic package Containers.Multiway_Trees is new.

Inconsistencies With Ada 2012

{*AI12-0111-1*} **Correction:** Tampering with elements is now defined to be equivalent to tampering with cursors for ordinary containers. If a program requires tampering detection to work, it might fail in Ada 2022. Needless to say, this shouldn't happen outside of test programs. See Inconsistencies With Ada 2012 in A.18.2 for more details. 233.d/5

Incompatibilities With Ada 2012

{*AI12-0111-1*} {*AI12-0112-1*} A number of new subprograms, types, and even a nested package were added to Containers.Multiway_Trees to better support contracts and stable views. Therefore, a use clause conflict is possible; see the introduction of Annex A for more on this topic. 233.e/5

Extensions to Ada 2012

{*AI12-0196-1*} **Correction:** Replace_Element is now defined such that it can be used concurrently so long as it operates on different elements. This allows some container operations to be used in parallel without separate synchronization. 233.f/5

{*AI12-0266-1*} Most iterators can now return parallel iterators, to be used in parallel constructs. 233.g/5

Wording Changes from Ada 2012

{*AI12-0069-1*} **Corrigendum:** Fixed the function Iterate so it is clear that the root node is never visited. 233.h/4

{*AI12-0078-1*} **Corrigendum:** The definition of *node* is clarified so that it it doesn't appear to say all nodes have an element. 233.i/4

{*AI12-0110-1*} **Corrigendum:** Clarified that tampering checks precede all other checks made by a subprogram (but come after those associated with the call). 233.j/4

{*AI12-0112-1*} Added contracts to this container. This includes describing some of the semantics with pre- and postconditions, rather than English text. Note that the preconditions can be Suppressed (see 11.5). 233.k/5

A.18.11 The Generic Package Containers.Indefinite_Vectors

{*AI95-00302-03*} The language-defined generic package Containers.Indefinite_Vectors provides a private type Vector and a set of operations. It provides the same operations as the package Containers.Vectors (see A.18.2), with the difference that the generic formal Element_Type is indefinite. 1/2

Static Semantics

{*AI95-00302-03*} {*AI05-0092-1*} The declaration of the generic library package Containers.Indefinite_Vectors has the same contents and semantics as Containers.Vectors except: 2/3

- The generic formal Element_Type is indefinite. 3/2
- The procedures with the profiles: 4/2

```
procedure Insert (Container : in out Vector;                    5/2
                  Before    : in     Extended_Index;
                  Count     : in     Count_Type := 1);
procedure Insert (Container : in out Vector;                    6/2
                  Before    : in     Cursor;
                  Position  :    out Cursor;
                  Count     : in     Count_Type := 1);
```

  are omitted. 7/2

  Discussion: These procedures are omitted because there is no way to create a default-initialized object of an indefinite type. Note that Insert_Space can be used instead of this routine in most cases. Omitting the routine completely allows any problems to be diagnosed by the compiler when converting from a definite to indefinite vector. 7.a/2

- The actual Element parameter of access subprogram Process of Update_Element may be constrained even if Element_Type is unconstrained. 8/2
- {*AI12-0035-1*} The operations "&", Append, Insert, Prepend, Replace_Element, and To_Vector that have a formal parameter of type Element_Type perform indefinite insertion (see A.18). 9/4

10/5 • {*AI12-0111-1*} {*AI12-0112-1*} The description of Tampering_With_Elements_Prohibited is replaced by:

11/5 Returns True if tampering with elements is prohibited for Container, and False otherwise.

11.a/5 **Reason:** Complete replacement of an element can cause its memory to be deallocated while another operation is holding onto a reference to it. That can't be allowed. However, a simple modification of (part of) an element is not a problem, so Update_Element does not cause a problem.

12/5 • {*AI12-0111-1*} {*AI12-0112-1*} Tampering_With_Cursors_Prohibited is replaced by Tampering_With_Elements_Prohibited in the postcondition for the operations Reference and Constant_Reference.

13/5 • {*AI12-0111-1*} The operations Replace_Element, Reverse_Elements, and Swap, and the nested generic unit Generic_Sorting are omitted from the nested package Stable.

Extensions to Ada 95

13.a/2 {*AI95-00302-03*} The generic package Containers.Indefinite_Vectors is new.

Inconsistencies With Ada 2012

13.b/4 {*AI12-0035-1*} **Corrigendum:** Defined some routines to "perform indefinite insertion". This could mean that some calls to those routines would now raise Program_Error where they previously worked. However, this is extremely unlikely, as it would require that the package was not implemented in Ada (an Ada allocator would raise Program_Error in these circumstances), and that a program inserted a more nested tagged type (or access discriminant) into a container, and then used that object before its type or discriminant went out of scope. All known implementations are implemented in Ada, so we believe there is no practical incompatibility. As such, we mention this only for completeness.

13.c/5 {*AI12-0111-1*} Defined the Iterator_View aspect, so that the stable view is used for container element iterators. This means that tampering with elements is prohibited during the entire loop, rather than tampering with cursors being prohibited during the loop, and tampering with elements being prohibited only during the lifetimes of references to the loop parameter. Thus, if a container element iterator does an operation that tampers with elements on the iterated container, that operation will fail a tampering check in Ada 2022 (and thus raise Program_Error), while it would have worked in Ada 2012 so long as the loop parameter is not involved. We believe this to be a dubious loop structure that should be rare. Note that this issue only occurs for the indefinite container form, the ordinary and bounded containers allow such operations at any time in Ada 2022.

A.18.12 The Generic Package Containers.Indefinite_Doubly_Linked_Lists

1/2 {*AI95-00302-03*} The language-defined generic package Containers.Indefinite_Doubly_Linked_Lists provides private types List and Cursor, and a set of operations for each type. It provides the same operations as the package Containers.Doubly_Linked_Lists (see A.18.3), with the difference that the generic formal Element_Type is indefinite.

Static Semantics

2/3 {*AI95-00302-03*} {*AI05-0092-1*} The declaration of the generic library package Containers.Indefinite_-Doubly_Linked_Lists has the same contents and semantics as Containers.Doubly_Linked_Lists except:

3/2 • The generic formal Element_Type is indefinite.

4/2 • The procedure with the profile:

5/2
```
procedure Insert (Container : in out List;
                  Before    : in     Cursor;
                  Position  :    out Cursor;
                  Count     : in     Count_Type := 1);
```

6/2 is omitted.

6.a/2 **Discussion:** This procedure is omitted because there is no way to create a default-initialized object of an indefinite type. We considered having this routine insert an empty element similar to the empty elements of a vector, but rejected

this possibility because the semantics are fairly complex and very different from the existing definite container. That would make it more error-prone to convert a container from a definite type to an indefinite type; by omitting the routine completely, any problems will be diagnosed by the compiler.

- The actual Element parameter of access subprogram Process of Update_Element may be constrained even if Element_Type is unconstrained. 7/2
- {*AI12-0035-1*} The operations Append, Insert, Prepend, and Replace_Element that have a formal parameter of type Element_Type perform indefinite insertion (see A.18). 8/4
- {*AI12-0111-1*} {*AI12-0112-1*} The description of Tampering_With_Elements_Prohibited is replaced by: 9/5

  Returns True if tampering with elements is prohibited for Container, and False otherwise. 10/5

  Reason: Complete replacement of an element can cause its memory to be deallocated while another operation is holding onto a reference to it. That can't be allowed. However, a simple modification of (part of) an element is not a problem. 10.a/5
- {*AI12-0111-1*} {*AI12-0112-1*} Tampering_With_Cursors_Prohibited is replaced by Tampering_With_Elements_Prohibited in the postcondition for the operations Reference and Constant_Reference. 11/5
- {*AI12-0111-1*} The operations Replace_Element and Swap are omitted from the nested package Stable. 12/5

Extensions to Ada 95

{*AI95-00302-03*} The generic package Containers.Indefinite_Doubly_Linked_Lists is new. 12.a/2

Inconsistencies With Ada 2012

{*AI12-0035-1*} **Corrigendum:** Defined some routines to "perform indefinite insertion". This could mean that some calls to those routines would now raise Program_Error where they previously worked. However, this is extremely unlikely; see Inconsistencies With Ada 2012 in A.18.11 for details. 12.b/4

{*AI12-0111-1*} Defined the Iterator_View aspect, so that the stable view is used for container element iterators. This leads to a rare situation where Program_Error will be raised in Ada 2022 for code that would have worked in Ada 2012. See Inconsistencies With Ada 2012 in A.18.11 for details. 12.c/5

A.18.13 The Generic Package Containers.Indefinite_Hashed_Maps

{*AI95-00302-03*} The language-defined generic package Containers.Indefinite_Hashed_Maps provides a map with the same operations as the package Containers.Hashed_Maps (see A.18.5), with the difference that the generic formal types Key_Type and Element_Type are indefinite. 1/2

Static Semantics

{*AI95-00302-03*} {*AI05-0092-1*} The declaration of the generic library package Containers.Indefinite_Hashed_Maps has the same contents and semantics as Containers.Hashed_Maps except: 2/3

- The generic formal Key_Type is indefinite. 3/2
- The generic formal Element_Type is indefinite. 4/2
- The procedure with the profile: 5/2

```
procedure Insert (Container : in out Map;                    6/2
                  Key       : in     Key_Type;
                  Position  :    out Cursor;
                  Inserted  :    out Boolean);
```

  is omitted. 7/2

  Discussion: This procedure is omitted because there is no way to create a default-initialized object of an indefinite type. We considered having this routine insert an empty element similar to the empty elements of a vector, but rejected 7.a/2

this possibility because the semantics are fairly complex and very different from the existing case. That would make it more error-prone to convert a container from a definite type to an indefinite type; by omitting the routine completely, any problems will be diagnosed by the compiler.

8/2 • The actual Element parameter of access subprogram Process of Update_Element may be constrained even if Element_Type is unconstrained.

9/4 • {*AI12-0035-1*} The operations Include, Insert, Replace, and Replace_Element that have a formal parameter of type Element_Type perform indefinite insertion (see A.18).

9.a/4 **Discussion:** Some of the named operations also have a formal of the indefinite formal type Key_Type and perform indefinite insertion using that value, but it is sufficient to mention the formal of type Element_Type to cover those.

10/5 • {*AI12-0111-1*} {*AI12-0112-1*} The description of Tampering_With_Elements_Prohibited is replaced by:

11/5 Returns True if tampering with elements is prohibited for Container, and False otherwise.

11.a/5 **Reason:** Complete replacement of an element can cause its memory to be deallocated while another operation is holding onto a reference to it. That can't be allowed. However, a simple modification of (part of) an element is not a problem.

12/5 • {*AI12-0111-1*} {*AI12-0112-1*} Tampering_With_Cursors_Prohibited is replaced by Tampering_With_Elements_Prohibited in the postcondition for the operations Reference and Constant_Reference.

13/5 • {*AI12-0111-1*} The operations Replace and Replace_Element are omitted from the nested package Stable.

Extensions to Ada 95

13.a/2 {*AI95-00302-03*} The generic package Containers.Indefinite_Hashed_Maps is new.

Inconsistencies With Ada 2012

13.b/4 {*AI12-0035-1*} **Corrigendum:** Defined some routines to “perform indefinite insertion”. This could mean that some calls to those routines would now raise Program_Error where they previously worked. However, this is extremely unlikely; see Inconsistencies With Ada 2012 in A.18.11 for details.

13.c/5 {*AI12-0111-1*} Defined the Iterator_View aspect, so that the stable view is used for container element iterators. This leads to a rare situation where Program_Error will be raised in Ada 2022 for code that would have worked in Ada 2012. See Inconsistencies With Ada 2012 in A.18.11 for details.

A.18.14 The Generic Package Containers.Indefinite_Ordered_Maps

1/2 {*AI95-00302-03*} The language-defined generic package Containers.Indefinite_Ordered_Maps provides a map with the same operations as the package Containers.Ordered_Maps (see A.18.6), with the difference that the generic formal types Key_Type and Element_Type are indefinite.

Static Semantics

2/3 {*AI95-00302-03*} {*AI05-0092-1*} The declaration of the generic library package Containers.Indefinite_Ordered_Maps has the same contents and semantics as Containers.Ordered_Maps except:

3/2 • The generic formal Key_Type is indefinite.

4/2 • The generic formal Element_Type is indefinite.

5/2 • The procedure with the profile:

6/2
```
procedure Insert (Container : in out Map;
                  Key       : in     Key_Type;
                  Position  :    out Cursor;
                  Inserted  :    out Boolean);
```

7/2 is omitted.

Discussion: This procedure is omitted because there is no way to create a default-initialized object of an indefinite type. We considered having this routine insert an empty element similar to the empty elements of a vector, but rejected this possibility because the semantics are fairly complex and very different from the existing case. That would make it more error-prone to convert a container from a definite type to an indefinite type; by omitting the routine completely, any problems will be diagnosed by the compiler. 7.a/2

- The actual Element parameter of access subprogram Process of Update_Element may be constrained even if Element_Type is unconstrained. 8/2
- {*AI12-0035-1*} The operations Include, Insert, Replace, and Replace_Element that have a formal parameter of type Element_Type perform indefinite insertion (see A.18). 9/4

  Discussion: Some of the named operations also have a formal of the indefinite formal type Key_Type and perform indefinite insertion using that value, but it is sufficient to mention the formal of type Element_Type to cover those. 9.a/4
- {*AI12-0111-1*} {*AI12-0112-1*} The description of Tampering_With_Elements_Prohibited is replaced by: 10/5

  Returns True if tampering with elements is prohibited for Container, and False otherwise. 11/5

  Reason: Complete replacement of an element can cause its memory to be deallocated while another operation is holding onto a reference to it. That can't be allowed. However, a simple modification of (part of) an element is not a problem. 11.a/5
- {*AI12-0111-1*} {*AI12-0112-1*} Tampering_With_Cursors_Prohibited is replaced by Tampering_With_Elements_Prohibited in the postcondition for the operations Reference and Constant_Reference. 12/5
- {*AI12-0111-1*} The operations Replace and Replace_Element are omitted from the nested package Stable. 13/5

Extensions to Ada 95

{*AI95-00302-03*} The generic package Containers.Indefinite_Ordered_Maps is new. 13.a/2

Inconsistencies With Ada 2012

{*AI12-0035-1*} **Corrigendum:** Defined some routines to “perform indefinite insertion”. This could mean that some calls to those routines would now raise Program_Error where they previously worked. However, this is extremely unlikely; see Inconsistencies With Ada 2012 in A.18.11 for details. 13.b/4

{*AI12-0111-1*} Defined the Iterator_View aspect, so that the stable view is used for container element iterators. This leads to a rare situation where Program_Error will be raised in Ada 2022 for code that would have worked in Ada 2012. See Inconsistencies With Ada 2012 in A.18.11 for details. 13.c/5

A.18.15 The Generic Package Containers.Indefinite_Hashed_Sets

{*AI95-00302-03*} The language-defined generic package Containers.Indefinite_Hashed_Sets provides a set with the same operations as the package Containers.Hashed_Sets (see A.18.8), with the difference that the generic formal type Element_Type is indefinite. 1/2

Static Semantics

{*AI95-00302-03*} {*AI05-0092-1*} The declaration of the generic library package Containers.Indefinite_Hashed_Sets has the same contents and semantics as Containers.Hashed_Sets except: 2/3

- The generic formal Element_Type is indefinite. 3/2
- The actual Element parameter of access subprogram Process of Update_Element_Preserving_Key may be constrained even if Element_Type is unconstrained. 4/2
- {*AI12-0035-1*} The operations Include, Insert, Replace, Replace_Element, and To_Set that have a formal parameter of type Element_Type perform indefinite insertion (see A.18). 5/4

5.a/4 **Ramification:** This includes the procedure Replace declared in the nested generic package Generic_Keys, as well as the routines declared directly in the Containers.Indefinite_Hashed_Sets package.

5.b/5 **Discussion:** {*AI12-0111-1*} Unlike the other containers, a Hashed_Set has no operations that tamper with elements without tampering with cursors. Modifying the contents of a element of a set can change its position in the set, so those operations tamper with cursors.

5.c/5 {*AI12-0111-1*} Because of this characteristic, this subclause is missing the rules that other indefinite containers have modifying the definition and use of Tampering_with_Elements, and the definition of the Stable subpackaage is the same as the definite version.

Extensions to Ada 95

5.d/2 {*AI95-00302-03*} The generic package Containers.Indefinite_Hashed_Sets is new.

Inconsistencies With Ada 2012

5.e/4 {*AI12-0035-1*} **Corrigendum:** Defined some routines to "perform indefinite insertion". This could mean that some calls to those routines would now raise Program_Error where they previously worked. However, this is extremely unlikely; see Inconsistencies With Ada 2012 in A.18.11 for details.

A.18.16 The Generic Package Containers.Indefinite_Ordered_Sets

1/2 {*AI95-00302-03*} The language-defined generic package Containers.Indefinite_Ordered_Sets provides a set with the same operations as the package Containers.Ordered_Sets (see A.18.9), with the difference that the generic formal type Element_Type is indefinite.

Static Semantics

2/3 {*AI95-00302-03*} {*AI05-0092-1*} The declaration of the generic library package Containers.Indefinite_Ordered_Sets has the same contents and semantics as Containers.Ordered_Sets except:

3/2 - The generic formal Element_Type is indefinite.

4/2 - The actual Element parameter of access subprogram Process of Update_Element_Preserving_Key may be constrained even if Element_Type is unconstrained.

5/4 - {*AI12-0035-1*} The operations Include, Insert, Replace, Replace_Element, and To_Set that have a formal parameter of type Element_Type perform indefinite insertion (see A.18).

5.a/4 **Ramification:** This includes the procedure Replace declared in the nested generic package Generic_Keys, as well as the routines declared directly in the Containers.Indefinite_Ordered_Sets package.

5.b/5 **Discussion:** {*AI12-0111-1*} Unlike the other containers, a Ordered_Set has no operations that tamper with elements without tampering with cursors. Modifying the contents of a element of a set can change its position in the set, so those operations tamper with cursors.

5.c/5 {*AI12-0111-1*} Because of this characteristic, this subclause is missing the rules that other indefinite containers have modifying the definition and use of Tampering_with_Elements, and the definition of the Stable subpackaage is the same as the definite version.

Extensions to Ada 95

5.d/2 {*AI95-00302-03*} The generic package Containers.Indefinite_Ordered_Sets is new.

Inconsistencies With Ada 2012

5.e/4 {*AI12-0035-1*} **Corrigendum:** Defined some routines to "perform indefinite insertion". This could mean that some calls to those routines would now raise Program_Error where they previously worked. However, this is extremely unlikely; see Inconsistencies With Ada 2012 in A.18.11 for details.

A.18.17 The Generic Package Containers.Indefinite_Multiway_Trees

{*AI05-0136-1*} The language-defined generic package Containers.Indefinite_Multiway_Trees provides a multiway tree with the same operations as the package Containers.Multiway_Trees (see A.18.10), with the difference that the generic formal Element_Type is indefinite. 1/3

Static Semantics

{*AI05-0136-1*} The declaration of the generic library package Containers.Indefinite_Multiway_Trees has the same contents and semantics as Containers.Multiway_Trees except: 2/3

- The generic formal Element_Type is indefinite. 3/3
- The procedure with the profile: 4/3

```
procedure Insert_Child (Container : in out Tree;                    5/3
                        Parent    : in     Cursor;
                        Before    : in     Cursor;
                        Position  :    out Cursor;
                        Count     : in     Count_Type := 1);
```

  is omitted. 6/3

  Discussion: This procedure is omitted because there is no way to create a default-initialized object of an indefinite type. We considered having this routine insert an empty element similar to the empty elements of a vector, but rejected this possibility because the semantics are fairly complex and very different from the existing case. That would make it more error-prone to convert a container from a definite type to an indefinite type; by omitting the routine completely, any problems will be diagnosed by the compiler. 6.a/3

- The actual Element parameter of access subprogram Process of Update_Element may be constrained even if Element_Type is unconstrained. 7/3
- {*AI12-0035-1*} The operations Append_Child, Insert_Child, Prepend_Child, and Replace_Element that have a formal parameter of type Element_Type perform indefinite insertion (see A.18). 8/4
- {*AI12-0111-1*} {*AI12-0112-1*} The description of Tampering_With_Elements_Prohibited is replaced by: 9/5

  Returns True if tampering with elements is prohibited for Container, and False otherwise. 10/5

  Reason: Complete replacement of an element can cause its memory to be deallocated while another operation is holding onto a reference to it. That can't be allowed. However, a simple modification of (part of) an element is not a problem. 10.a/5
- {*AI12-0111-1*} {*AI12-0112-1*} Tampering_With_Cursors_Prohibited is replaced by Tampering_With_Elements_Prohibited in the postcondition for the operations Reference and Constant_Reference. 11/5
- {*AI12-0111-1*} The operations Replace_Element and Swap are omitted from the nested package Stable. 12/5

Extensions to Ada 2005

{*AI05-0136-1*} The generic package Containers.Indefinite_Multiway_Trees is new. 12.a/3

Inconsistencies With Ada 2012

{*AI12-0035-1*} **Corrigendum:** Defined some routines to “perform indefinite insertion”. This could mean that some calls to those routines would now raise Program_Error where they previously worked. However, this is extremely unlikely; see Inconsistencies With Ada 2012 in A.18.11 for details. 12.b/4

{*AI12-0111-1*} Defined the Iterator_View aspect, so that the stable view is used for container element iterators. This leads to a rare situation where Program_Error will be raised in Ada 2022 for code that would have worked in Ada 2012. See Inconsistencies With Ada 2012 in A.18.11 for details. 12.c/5

A.18.18 The Generic Package Containers.Indefinite_Holders

1/3 {*AI05-0069-1*} The language-defined generic package Containers.Indefinite_Holders provides a private type Holder and a set of operations for that type. A holder container holds a single element of an indefinite type.

2/3 {*AI05-0069-1*} A holder container allows the declaration of an object that can be used like an uninitialized variable or component of an indefinite type.

3/3 {*AI05-0069-1*} A holder container may be *empty*. An empty holder does not contain an element.

Static Semantics

4/3 {*AI05-0069-1*} The generic library package Containers.Indefinite_Holders has the following declaration:

5/5
```
{AI05-0069-1} {AI05-0084-1} {AI12-0112-1} generic
   type Element_Type (<>) is private;
   with function "=" (Left, Right : Element_Type) return Boolean is <>;
package Ada.Containers.Indefinite_Holders
   with Preelaborate, Remote_Types,
        Nonblocking, Global => in out synchronized is
```

5.a/5 **Discussion:** {*AI12-0112-1*} For discussion on the reasons and meaning of the specifications of the Global and Nonblocking aspects in this generic package, see the notes on the equivalent operations in the specification of the Containers.Vectors package (see A.18.2).

6/5
```
   {AI12-0112-1} {AI12-0399-1}    type Holder is tagged private
         with Stable_Properties => (Is_Empty,
                                    Tampering_With_The_Element_Prohibited),
              Default_Initial_Condition => Is_Empty (Holder),
              Preelaborable_Initialization;
```

7/3
```
      Empty_Holder : constant Holder;
```

7.1/5
```
   {AI12-0112-1}    function Equal_Element (Left, Right : Element_Type) return Boolean
         renames "=";
```

8/3
```
      function "=" (Left, Right : Holder) return Boolean;
```

8.1/5
```
   {AI12-0112-1}    function Tampering_With_The_Element_Prohibited
        (Container : Holder) return Boolean
         with Nonblocking, Global => null, Use_Formal => null;
```

8.2/5
```
   {AI12-0339-1}    function Empty return Holder
         is (Empty_Holder)
         with Post =>
               not Tampering_With_The_Element_Prohibited (Empty'Result)
               and then Is_Empty (Empty'Result);
```

9/5
```
   {AI12-0112-1}    function To_Holder (New_Item : Element_Type) return Holder
         with Post => not Is_Empty (To_Holder'Result);
```

10/5
```
   {AI12-0112-1}    function Is_Empty (Container : Holder) return Boolean
         with Global => null, Use_Formal => null;
```

11/5
```
   {AI12-0112-1}    procedure Clear (Container : in out Holder)
         with Pre  => not Tampering_With_The_Element_Prohibited (Container)
                      or else raise Program_Error,
              Post => Is_Empty (Container);
```

12/5
```
   {AI12-0112-1}    function Element (Container : Holder) return Element_Type
         with Pre => not Is_Empty (Container) or else raise Constraint_Error,
              Global => null, Use_Formal => Element_Type;
```

13/5
```
   {AI12-0112-1}    procedure Replace_Element (Container : in out Holder;
                                               New_Item  : in     Element_Type)
         with Pre  => not Tampering_With_The_Element_Prohibited (Container)
                      or else raise Program_Error,
              Post => not Is_Empty (Container);
```

```
{AI12-0112-1}    procedure Query_Element                                                    14/5
     (Container : in Holder;
      Process   : not null access procedure (Element : in Element_Type))
      with Pre => not Is_Empty (Container) or else raise Constraint_Error;

{AI05-0069-1} {AI05-0248-1} {AI12-0112-1}    procedure Update_Element                    15/5
     (Container : in out Holder;
      Process   : not null access procedure (Element : in out Element_Type))
      with Pre => not Is_Empty (Container) or else raise Constraint_Error;

{AI05-0212-1} {AI12-0112-1}    type Constant_Reference_Type                               16/5
         (Element : not null access constant Element_Type) is private
      with Implicit_Dereference => Element,
           Nonblocking, Global => in out synchronized,
           Default_Initial_Condition => (raise Program_Error);

{AI05-0212-1} {AI12-0112-1}    type Reference_Type                                        17/5
         (Element : not null access Element_Type) is private
      with Implicit_Dereference => Element,
           Nonblocking, Global => in out synchronized,
           Default_Initial_Condition => (raise Program_Error);

{AI05-0212-1} {AI12-0112-1}    function Constant_Reference (Container : aliased in       18/5
Holder)
      return Constant_Reference_Type
      with Pre  => not Is_Empty (Container)
                     or else raise Constraint_Error,
           Post => Tampering_With_The_Element_Prohibited (Container),
           Nonblocking, Global => null, Use_Formal => null;

{AI05-0212-1} {AI12-0112-1}    function Reference (Container : aliased in out            19/5
Holder)
      return Reference_Type
      with Pre  => not Is_Empty (Container)
                     or else raise Constraint_Error,
           Post => Tampering_With_The_Element_Prohibited (Container),
           Nonblocking, Global => null, Use_Formal => null;

{AI05-0001-1} {AI12-0112-1}    procedure Assign (Target : in out Holder; Source :        20/5
in Holder)
      with Post => (Is_Empty (Source) = Is_Empty (Target));

{AI05-0001-1} {AI12-0112-1}    function Copy (Source : Holder) return Holder             21/5
      with Post => (Is_Empty (Source) = Is_Empty (Copy'Result));

{AI12-0112-1}    procedure Move (Target : in out Holder; Source : in out Holder)          22/5
      with Pre  => (not Tampering_With_The_Element_Prohibited (Target)
                     or else raise Program_Error) and then
                   (not Tampering_With_The_Element_Prohibited (Source)
                     or else raise Program_Error),
           Post => (if not Target'Has_Same_Storage (Source) then
                     Is_Empty (Source) and then (not Is_Empty (Target)));

   procedure Swap (Left, Right : in out Holder)                                           22.1/5
      with Pre  => (not Tampering_With_The_Element_Prohibited (Left)
                     or else raise Program_Error) and then
                   (not Tampering_With_The_Element_Prohibited (Right)
                     or else raise Program_Error),
           Post => Is_Empty (Left) = Is_Empty (Right)'Old and then
                   Is_Empty (Right) = Is_Empty (Left)'Old;

private                                                                                   23/3

   ... -- not specified by the language                                                   24/3

end Ada.Containers.Indefinite_Holders;                                                    25/3
```

{*AI05-0069-1*} The actual function for the generic formal function "=" on Element_Type values is expected to define a reflexive and symmetric relationship and return the same result value each time it is called with a particular pair of values. If it behaves in some other manner, the function "=" on holder values returns an unspecified value. The exact arguments and number of calls of this generic formal function by the function "=" on holder values are unspecified. 26/3

26.a/3 **Ramification:** If the actual function for "=" is not symmetric and consistent, the result returned by any of the functions defined to use "=" cannot be predicted. The implementation is not required to protect against "=" raising an exception, or returning random results, or any other "bad" behavior. And it can call "=" in whatever manner makes sense. But note that only the results of the function "=" is unspecified; other subprograms are not allowed to break if "=" is bad.

27/3 {*AI05-0069-1*} The type Holder is used to represent holder containers. The type Holder needs finalization (see 7.6).

28/3 {*AI05-0069-1*} Empty_Holder represents an empty holder object. If an object of type Holder is not otherwise initialized, it is initialized to the same value as Empty_Holder.

29/5 {*AI05-0069-1*} {*AI05-0262-1*} {*AI12-0112-1*} [Some operations check for “tampering with the element” of a container because they depend on the element of the container not being replaced.] When tampering with the element is *prohibited* for a particular holder object *H*, Program_Error is propagated by the finalization of *H*[, as well as by a call that passes *H* to certain of the operations of this package, as indicated by the precondition of such an operation].

Paragraphs 30 through 35 are removed as preconditions now describe these rules.

34.a/3 **Reason:** Complete replacement of an element can cause its memory to be deallocated while another operation is holding onto a reference to it. That can't be allowed. However, a simple modification of (part of) an element is not a problem, so Update_Element does not cause a problem.

36/3
```
function "=" (Left, Right : Holder) return Boolean;
```

37/3 {*AI05-0069-1*} If Left and Right denote the same holder object, then the function returns True. Otherwise, it compares the element contained in Left to the element contained in Right using the generic formal equality operator, returning the result of that operation. Any exception raised during the evaluation of element equality is propagated.

37.a/3 **Implementation Note:** This wording describes the canonical semantics. However, the order and number of calls on the formal equality **function** is unspecified, so an implementation need not call the equality function if the correct answer can be determined without doing so.

37.1/5
```
{AI12-0112-1} function Tampering_With_The_Element_Prohibited
  (Container : Holder) return Boolean
   with Nonblocking, Global => null, Use_Formal => null;
```

37.2/5 {*AI12-0112-1*} Returns True if tampering with the element is currently prohibited for Container, and returns False otherwise.

38/5
```
{AI12-0112-1} function To_Holder (New_Item : Element_Type) return Holder
   with Post => not Is_Empty (To_Holder'Result);
```

39/4 {*AI05-0069-1*} {*AI12-0035-1*} Returns a nonempty holder containing an element initialized to New_Item. To_Holder performs indefinite insertion (see A.18).

40/5
```
{AI12-0112-1} function Is_Empty (Container : Holder) return Boolean
   with Global => null, Use_Formal => null;
```

41/3 {*AI05-0069-1*} Returns True if Container is empty, and False if it contains an element.

42/5
```
{AI12-0112-1} procedure Clear (Container : in out Holder)
   with Pre  => not Tampering_With_The_Element_Prohibited (Container)
                or else raise Program_Error,
        Post => Is_Empty (Container);
```

43/5 {*AI05-0069-1*} {*AI12-0112-1*} Removes the element from Container.

44/5
```
{AI12-0112-1} function Element (Container : Holder) return Element_Type
   with Pre => not Is_Empty (Container) or else raise Constraint_Error,
        Global => null, Use_Formal => Element_Type;
```

45/5 {*AI05-0069-1*} {*AI12-0112-1*} Returns the element stored in Container.

```
{AI12-0112-1} procedure Replace_Element (Container : in out Holder;                    46/5
                                         New_Item  : in     Element_Type)
   with Pre  => not Tampering_With_The_Element_Prohibited (Container)
                 or else raise Program_Error,
        Post => not Is_Empty (Container);
```

{*AI05-0069-1*} {*AI12-0035-1*} {*AI12-0112-1*} Replace_Element assigns the value New_Item into Container, replacing any preexisting content of Container; Replace_Element performs indefinite insertion (see A.18). 47/5

```
{AI12-0112-1} procedure Query_Element                                                  48/5
  (Container : in Holder;
   Process   : not null access procedure (Element : in Element_Type))
   with Pre => not Is_Empty (Container) or else raise Constraint_Error,
        Global => null, Use_Formal => null;
```

{*AI05-0069-1*} {*AI05-0262-1*} {*AI05-0265-1*} {*AI12-0112-1*} Query_Element calls Process.**all** with the contained element as the argument. Tampering with the element of Container is prohibited during the execution of the call on Process.**all**. Any exception raised by Process.**all** is propagated. 49/5

Implementation Note: {*AI05-0005-1*} The "tamper with the element" check is intended to prevent the Element parameter of Process from being replaced or deleted outside of Process. The check prevents data loss (if Element_Type is passed by copy) or erroneous execution (if Element_Type is an unconstrained type). 49.a/3

```
{AI05-0069-1} {AI05-0248-1} {AI12-0112-1} procedure Update_Element                     50/5
  (Container : in out Holder;
   Process   : not null access procedure (Element : in out Element_Type))
   with Pre => not Is_Empty (Container) or else raise Constraint_Error;
```

{*AI05-0069-1*} {*AI05-0262-1*} {*AI05-0265-1*} {*AI12-0112-1*} Update_Element calls Process.**all** with the contained element as the argument. Tampering with the element of Container is prohibited during the execution of the call on Process.**all**. Any exception raised by Process.**all** is propagated. 51/5

Implementation Note: The Element parameter of Process.**all** may be constrained even if Element_Type is unconstrained. 51.a/3

```
{AI05-0212-1} {AI12-0112-1} type Constant_Reference_Type                               52/5
      (Element : not null access constant Element_Type) is private
   with Implicit_Dereference => Element,
        Nonblocking, Global => in out synchronized,
        Default_Initial_Condition => (raise Program_Error);

{AI05-0212-1} {AI12-0112-1} type Reference_Type (Element : not null access             53/5
Element_Type) is private
   with Implicit_Dereference => Element,
        Nonblocking, Global => in out synchronized,
        Default_Initial_Condition => (raise Program_Error);
```

{*AI05-0212-1*} The types Constant_Reference_Type and Reference_Type need finalization. 54/3

This paragraph was deleted.{*AI05-0212-1*} {*AI12-0112-1*} 55/5

Reason: It is expected that Reference_Type (and Constant_Reference_Type) will be a controlled type, for which finalization will have some action to terminate the tampering check for the associated container. If the object is created by default, however, there is no associated container. Since this is useless, and supporting this case would take extra work, we define it to raise an exception. 55.a/3

56/5
```
{AI05-0212-1} {AI12-0112-1} function Constant_Reference (Container : aliased in
Holder)
   return Constant_Reference_Type
   with Pre  => not Is_Empty (Container) or else raise Constraint_Error,
        Post => Tampering_With_The_Element_Prohibited (Container),
        Nonblocking, Global => null, Use_Formal => null;
```

57/3 {*AI05-0212-1*} This function (combined with the Implicit_Dereference aspect) provides a convenient way to gain read access to the contained element of a holder container.

58/5 {*AI05-0212-1*} {*AI05-0262-1*} {*AI05-0265-1*} {*AI12-0112-1*} Constant_Reference returns an object whose discriminant is an access value that designates the contained element. Tampering with the element of Container is prohibited while the object returned by Constant_Reference exists and has not been finalized.

59/5
```
{AI05-0212-1} {AI12-0112-1} function Reference (Container : aliased in out Holder)
   return Reference_Type
   with Pre  => not Is_Empty (Container) or else raise Constraint_Error,
        Post => Tampering_With_The_Element_Prohibited (Container),
        Nonblocking, Global => null, Use_Formal => null;
```

60/3 {*AI05-0212-1*} This function (combined with the Implicit_Dereference aspects) provides a convenient way to gain read and write access to the contained element of a holder container.

61/5 {*AI05-0212-1*} {*AI05-0262-1*} {*AI05-0265-1*} {*AI12-0112-1*} Reference returns an object whose discriminant is an access value that designates the contained element. Tampering with the element of Container is prohibited while the object returned by Reference exists and has not been finalized.

62/5
```
{AI12-0112-1} procedure Assign (Target : in out Holder; Source : in Holder)
   with Post => (Is_Empty (Source) = Is_Empty (Target));
```

63/3 {*AI05-0001-1*} If Target denotes the same object as Source, the operation has no effect. If Source is empty, Clear (Target) is called. Otherwise, Replace_Element (Target, Element (Source)) is called.

63.a/3 **Discussion:** {*AI05-0005-1*} This routine exists for compatibility with the other containers. For a holder, `Assign(A, B)` and `A := B` behave effectively the same. (Assign Clears the Target, while := finalizes the Target, but these should have similar effects.)

64/5
```
{AI12-0112-1} function Copy (Source : Holder) return Holder
   with Post => (Is_Empty (Source) = Is_Empty (Copy'Result));
```

65/3 {*AI05-0001-1*} If Source is empty, returns an empty holder container; otherwise, returns To_Holder (Element (Source)).

66/5
```
{AI12-0112-1} procedure Move (Target : in out Holder; Source : in out Holder)
   with Pre  => (not Tampering_With_The_Element_Prohibited (Target)
                    or else raise Program_Error) and then
                (not Tampering_With_The_Element_Prohibited (Source)
                    or else raise Program_Error),
        Post => (if not Target'Has_Same_Storage (Source) then
                    Is_Empty (Source) and then (not Is_Empty (Target)));
```

67/5 {*AI05-0069-1*} {*AI05-0248-1*} {*AI12-0112-1*} If Target denotes the same object as Source, then the operation has no effect. Otherwise, the element contained by Source (if any) is removed from Source and inserted into Target, replacing any preexisting content.

```
procedure Swap (Left, Right : in out Holder)
   with Pre  => (not Tampering_With_The_Element_Prohibited (Left)
                    or else raise Program_Error) and then
                (not Tampering_With_The_Element_Prohibited (Right)
                    or else raise Program_Error),
        Post => Is_Empty (Left) = Is_Empty (Right)'Old and then
                Is_Empty (Right) = Is_Empty (Left)'Old;
```

67.1/5

{*AI12-0350-1*} If Left denotes the same object as Right, then the operation has no effect. Otherwise, operation exchanges the elements (if any) contained by Left and Right. 67.2/5

Bounded (Run-Time) Errors

{*AI05-0022-1*} {*AI05-0069-1*} {*AI05-0248-1*} {*AI05-0262-1*} It is a bounded error for the actual function associated with a generic formal subprogram, when called as part of an operation of this package, to tamper with the element of any Holder parameter of the operation. Either Program_Error is raised, or the operation works as defined on the value of the Holder either prior to, or subsequent to, some or all of the modifications to the Holder. 68/3

{*AI05-0027-1*} {*AI05-0069-1*} It is a bounded error to call any subprogram declared in the visible part of Containers.Indefinite_Holders when the associated container has been finalized. If the operation takes Container as an **in out** parameter, then it raises Constraint_Error or Program_Error. Otherwise, the operation either proceeds as it would for an empty container, or it raises Constraint_Error or Program_Error. 69/3

Erroneous Execution

{*AI05-0212-1*} {*AI05-0269-1*} Execution is erroneous if the holder container associated with the result of a call to Reference or Constant_Reference is finalized before the result object returned by the call to Reference or Constant_Reference is finalized. 70/3

Reason: {*AI05-0212-1*} Each object of Reference_Type and Constant_Reference_Type probably contains some reference to the originating container. If that container is prematurely finalized (which is only possible via Unchecked_Deallocation, as accessibility checks prevent passing a container to Reference that will not live as long as the result), the finalization of the object of Reference_Type will try to access a nonexistent object. This is a normal case of a dangling pointer created by Unchecked_Deallocation; we have to explicitly mention it here as the pointer in question is not visible in the specification of the type. (This is the same reason we have to say this for invalid cursors.) 70.a/3

Implementation Requirements

{*AI05-0069-1*} No storage associated with a holder object shall be lost upon assignment or scope exit. 71/3

{*AI05-0069-1*} {*AI05-0269-1*} The execution of an assignment_statement for a holder container shall have the effect of copying the element (if any) from the source holder object to the target holder object. 72/3

Implementation Note: {*AI05-0298-1*} An assignment of a holder container is a "deep" copy; that is the element is copied as well as any data structures. We say "effect of" in order to allow the implementation to avoid copying the element immediately if it wishes. For instance, an implementation that avoided copying until one of the containers is modified would be allowed. (Note that this implementation would require care, see A.18.2 for more.) 72.a/3

Implementation Advice

{*AI05-0069-1*} {*AI05-0269-1*} {*AI12-0350-1*} Move and Swap should not copy any elements, and should minimize copying of internal data structures. 73/5

Implementation Advice: Move and Swap in Containers.Indefinite_Holders should not copy any elements, and should minimize copying of internal data structures. 73.a.1/5

Implementation Note: {*AI12-0350-1*} Usually that can be accomplished simply by moving the pointer(s) to the internal data structures appropriately. 73.a/5

{*AI05-0069-1*} {*AI05-0269-1*} If an exception is propagated from a holder operation, no storage should be lost, nor should the element be removed from a holder container unless specified by the operation. 74/3

74.a.1/3 **Implementation Advice:** If an exception is propagated from a holder operation, no storage should be lost, nor should the element be removed from a holder container unless specified by the operation.

74.a/3 **Reason:** This is important so that programs can recover from errors. But we don't want to require heroic efforts, so we just require documentation of cases where this can't be accomplished.

Extensions to Ada 2005

74.b/3 {*AI05-0069-1*} {*AI05-0084-1*} {*AI05-0265-1*} The generic package Containers.Indefinite_Holders is new.

Inconsistencies With Ada 2012

74.c/4 {*AI12-0035-1*} **Corrigendum:** Defined some routines to "perform indefinite insertion". This could mean that some calls to those routines would now raise Program_Error where they previously worked. However, this is extremely unlikely; see Inconsistencies With Ada 2012 in A.18.11 for details.

Incompatibilities With Ada 2012

74.d/5 {*AI12-0111-1*} {*AI12-0112-1*} {*AI12-0350-1*} A number of new subprograms and types were added to Containers.Indefinite_Holders to better support contracts and provide additional functionality. Therefore, a use clause conflict is possible; see the introduction of Annex A for more on this topic.

Wording Changes from Ada 2012

74.e/4 {*AI12-0110-1*} **Corrigendum:** Clarified that tampering checks precede all other checks made by a subprogram (but come after those associated with the call).

74.f/5 {*AI12-0112-1*} Added contracts to this container. This includes describing some of the semantics with pre- and postconditions, rather than English text. Note that the preconditions can be Suppressed (see 11.5).

A.18.19 The Generic Package Containers.Bounded_Vectors

1/3 {*AI05-0001-1*} The language-defined generic package Containers.Bounded_Vectors provides a private type Vector and a set of operations. It provides the same operations as the package Containers.Vectors (see A.18.2), with the difference that the maximum storage is bounded.

Static Semantics

2/3 {*AI05-0001-1*} The declaration of the generic library package Containers.Bounded_Vectors has the same contents and semantics as Containers.Vectors except:

3/5 - {*AI12-0112-1*} The aspect Preelaborate is replaced with aspect Pure. Aspect Global is deleted.

3.a/5 **Ramification:** {*AI12-0112-1*} The Global aspect for a Pure package is **null** (see 6.1.2), so we don't need to give it explicitly.

4/3 - The type Vector is declared with a discriminant that specifies the capacity:

5/5 {*AI12-0112-1*}
```
type Vector (Capacity : Count_Type) is tagged private...
```

5.1/5 - {*AI12-0409-1*} The aspect_definition for Preelaborable_Initialization for type Vector is changed to:

5.2/5
```
Preelaborable_Initialization =>
   Element_Type'Preelaborable_Initialization
```

6/3 - The type Vector needs finalization if and only if type Element_Type needs finalization.

6.a/3 **Implementation Note:** {*AI05-0212-1*} The type Vector cannot depend on package Ada.Finalization unless the element type depends on that package. The objects returned from the Iterator and Reference functions probably do depend on package Ada.Finalization. Restricted environments may need to avoid use of those functions and their associated types.

6.1/5 - {*AI12-0112-1*} Capacity is omitted from the Stable_Properties of type Vector.

6.b/5 **Reason:** The capacity is a discriminant here, so it can't be changed by most routines; thus including it in the stable properties adds no information.

6.2/5 - {*AI12-0339-1*} In function Empty, the postcondition is altered to:

```
Post =>
   Empty'Result.Capacity = Capacity and then
   not Tampering_With_Elements_Prohibited (Empty'Result) and then
   not Tampering_With_Cursors_Prohibited (Empty'Result) and then
   Length (Empty'Result) = 0;
```

6.3/5

- {*AI12-0112-1*} In function Copy, the postcondition is altered to: 7/5

```
Post => Length (Copy'Result) = Length (Source) and then
        (if Capacity > Length (Source) then
           Copy'Result.Capacity = Capacity
         else Copy'Result.Capacity >= Length (Source));
```

7.1/5

- The description of Reserve_Capacity is replaced with: 8/3

```
{AI12-0112-1}    procedure Reserve_Capacity (Container : in out Vector;
                                             Capacity  : in     Count_Type)
   with Pre => Capacity <= Container.Capacity
                  or else raise Capacity_Error;
```

8.1/5

{*AI12-0112-1*} This operation has no effect, [other than checking the precondition]. 9/5

- {*AI12-0112-1*} The portion of the postcondition checking the capacity is omitted from subprograms Set_Length, Assign, Insert, Insert_Space, Prepend, Append, and Delete. 9.1/5

- {*AI12-0112-1*} For procedures Insert, Insert_Space, Prepend, and Append, the part of the precondition reading: 9.2/5

```
(<some length> <= Maximum_Length - <some other length>
   or else raise Constraint_Error)
```

9.3/5

is replaced by: 9.4/5

```
(<some length> <= Maximum_Length - <some other length>
   or else raise Constraint_Error) and then
(<some length> <= Container.Capacity - <some other length>
   or else raise Capacity_Error)
```

9.5/5

Bounded (Run-Time) Errors

{*AI05-0160-1*} {*AI05-0265-1*} It is a bounded error to assign from a bounded vector object while tampering with elements [or cursors] of that object is prohibited. Either Program_Error is raised by the assignment, execution proceeds with the target object prohibiting tampering with elements [or cursors], or execution proceeds normally. 10/3

Proof: Tampering with elements includes tampering with cursors, so we only really need to talk about tampering with elements here; we mention cursors for clarity. 10.a/3

Erroneous Execution

{*AI05-0265-1*} When a bounded vector object *V* is finalized, if tampering with cursors is prohibited for *V* other than due to an assignment from another vector, then execution is erroneous. 11/3

Reason: This is a tampering event, but since the implementation is not allowed to use Ada.Finalization, it is not possible in a pure Ada implementation to detect this error. (There is no Finalize routine that will be called that could make the check.) Since the check probably cannot be made, the bad effects that could occur (such as an iterator going into an infinite loop or accessing a nonexistent element) cannot be prevented and we have to allow anything. We do allow re-assigning an object that only prohibits tampering because it was copied from another object as that cannot cause any negative effects. 11.a/3

Implementation Requirements

{*AI05-0184-1*} {*AI05-0264-1*} For each instance of Containers.Vectors and each instance of Containers.Bounded_Vectors, if the two instances meet the following conditions, then the output generated by the Vector'Output or Vector'Write subprograms of either instance shall be readable by the Vector'Input or Vector'Read of the other instance, respectively: 12/3

13/3 • {*AI05-0184-1*} {*AI05-0248-1*} the Element_Type parameters of the two instances are statically matching subtypes of the same type; and

14/3 • {*AI05-0184-1*} the output generated by Element_Type'Output or Element_Type'Write is readable by Element_Type'Input or Element_Type'Read, respectively (where Element_Type denotes the type of the two actual Element_Type parameters); and

15/3 • {*AI05-0184-1*} the preceding two conditions also hold for the Index_Type parameters of the instances.

Implementation Advice

16/3 {*AI05-0001-1*} Bounded vector objects should be implemented without implicit pointers or dynamic allocation.

16.a.1/3 **Implementation Advice:** Bounded vector objects should be implemented without implicit pointers or dynamic allocation.

17/3 {*AI05-0001-1*} The implementation advice for procedure Move to minimize copying does not apply.

17.a.1/3 **Implementation Advice:** The implementation advice for procedure Move to minimize copying does not apply to bounded vectors.

Extensions to Ada 2005

17.a/3 {*AI05-0001-1*} {*AI05-0160-1*} {*AI05-0184-1*} The generic package Containers.Bounded_Vectors is new.

Inconsistencies With Ada 2012

17.b/5 {*AI12-0111-1*} **Correction:** Tampering with elements is now defined to be equivalent to tampering with cursors for bounded containers. If a program requires tampering detection to work, it might fail in Ada 2022. Needless to say, this shouldn't happen outside of test programs. See Inconsistencies With Ada 2012 in A.18.2 for more details.

Incompatibilities With Ada 2012

17.c/5 {*AI12-0409-1*} **Correction:**A bounded vector now only has Preelaborable_Initialization (abbreviated PI in this note) when the actual for the Element_Type has PI. If an program used a vector whose actual Element_Type does not have PI in a context when PI is required (such as a library-level object in a preelaborated unit or as a component of a type with PI), the program would be illegal in Ada 2022 but legal in original Ada 2012. This situation is unlikely, especially as some existing Ada 2012 implementations reject the instance in this case.

A.18.20 The Generic Package Containers.Bounded_Doubly_Linked_Lists

1/3 {*AI05-0001-1*} The language-defined generic package Containers.Bounded_Doubly_Linked_Lists provides a private type List and a set of operations. It provides the same operations as the package Containers.Doubly_Linked_Lists (see A.18.3), with the difference that the maximum storage is bounded.

Static Semantics

2/3 {*AI05-0001-1*} The declaration of the generic library package Containers.Bounded_Doubly_Linked_Lists has the same contents and semantics as Containers.Doubly_Linked_Lists except:

3/5 • {*AI12-0112-1*} The aspect Preelaborate is replaced with aspect Pure. Aspect Global is deleted.

3.a/5 **Ramification:** {*AI12-0112-1*} The Global aspect for a Pure package is **null** (see 6.1.2), so we don't need to give it explicitly.

4/3 • The type List is declared with a discriminant that specifies the capacity (maximum number of elements) as follows:

5/5 {*AI12-0112-1*} `type List (Capacity : Count_Type) is tagged private...`

5.1/5 • {*AI12-0409-1*} The aspect_definition for Preelaborable_Initialization for type List is changed to:

```
Preelaborable_Initialization =>
   Element_Type'Preelaborable_Initialization
```
5.2/5

- The type List needs finalization if and only if type Element_Type needs finalization. 6/3

  Implementation Note: {*AI05-0212-1*} The type List cannot depend on package Ada.Finalization unless the element type depends on that package. The objects returned from the Iterator and Reference functions probably do depend on package Ada.Finalization. Restricted environments may need to avoid use of those functions and their associated types. 6.a/3

- {*AI12-0339-1*} The function Empty is replaced by: 6.1/5

```
function Empty (Capacity : Count_Type := implementation-defined)
   return List
   with Post =>
         Empty'Result.Capacity = Capacity and then
         not Tampering_With_Elements_Prohibited (Empty'Result) and then
         not Tampering_With_Cursors_Prohibited (Empty'Result) and then
         Length (Empty'Result) = 0;
```
6.2/5

- {*AI12-0112-1*} For procedures Insert, Prepend, Append, Merge, and the three-parameter Splice whose parameter Source has type List, the part of the precondition reading: 7/5

```
(<some length> <= Count_Type'Last - <some other length>
   or else raise Constraint_Error)
```
7.1/5

  is replaced by: 7.2/5

```
(<some length> <= Count_Type'Last - <some other length>
   or else raise Constraint_Error) and then
(<some length> <= Container.Capacity - <some other length>
   or else raise Capacity_Error)
```
7.3/5

- {*AI12-0112-1*} In procedure Assign, the precondition is altered to: 8/5

```
Pre => (not Tampering_With_Cursors_Prohibited (Target)
          or else raise Program_Error) and then
       (Length (Source) <= Target.Capacity
          or else raise Capacity_Error),
```
8.1/5

- The function Copy is replaced with: 9/3

{*AI12-0112-1*}
```
function Copy (Source : List; Capacity : Count_Type := 0)
   return List
   with Pre  => Capacity = 0 or else Capacity >= Length (Source)
                   or else raise Capacity_Error,
        Post =>
           Length (Copy'Result) = Length (Source) and then
           not Tampering_With_Elements_Prohibited (Copy'Result) and then
           not Tampering_With_Cursors_Prohibited (Copy'Result) and then
           Copy'Result.Capacity = (if Capacity = 0 then
              Length (Source) else Capacity);
```
10/5

  Returns a list whose elements have the same values as the elements of Source. 11/5

- *This paragraph was deleted.*{*AI12-0112-1*} 12/5

- {*AI12-0112-1*} In the four-parameter procedure Splice, the precondition is altered to: 13/5

13.1/5
```
Pre  => (not Tampering_With_Cursors_Prohibited (Target)
            or else raise Program_Error) and then
         (not Tampering_With_Cursors_Prohibited (Source)
            or else raise Program_Error) and then
         (Position /= No_Element
            or else raise Constraint_Error) and then
         (Has_Element (Source, Position)
            or else raise Program_Error) and then
         (Before = No_Element or else Has_Element (Target, Before)
            or else raise Program_Error) and then
         (Target'Has_Same_Storage (Source) or else
          Length (Target) /= Count_Type'Last
            or else raise Constraint_Error) and then
         (Target'Has_Same_Storage (Source) or else
          Length (Target) /= Target.Capacity
            or else raise Capacity_Error),
```

Bounded (Run-Time) Errors

14/3 {*AI05-0160-1*} {*AI05-0265-1*} It is a bounded error to assign from a bounded list object while tampering with elements [or cursors] of that object is prohibited. Either Program_Error is raised by the assignment, execution proceeds with the target object prohibiting tampering with elements [or cursors], or execution proceeds normally.

14.a/3 **Proof:** Tampering with elements includes tampering with cursors, so we only really need to talk about tampering with elements here; we mention cursors for clarity.

Erroneous Execution

15/3 {*AI05-0265-1*} When a bounded list object *L* is finalized, if tampering with cursors is prohibited for *L* other than due to an assignment from another list, then execution is erroneous.

15.a/3 **Reason:** This is a tampering event, but since the implementation is not allowed to use Ada.Finalization, it is not possible in a pure Ada implementation to detect this error. (There is no Finalize routine that will be called that could make the check.) Since the check probably cannot be made, the bad effects that could occur (such as an iterator going into an infinite loop or accessing a nonexistent element) cannot be prevented and we have to allow anything. We do allow re-assigning an object that only prohibits tampering because it was copied from another object as that cannot cause any negative effects.

Implementation Requirements

16/3 {*AI05-0184-1*} {*AI05-0264-1*} For each instance of Containers.Doubly_Linked_Lists and each instance of Containers.Bounded_Doubly_Linked_Lists, if the two instances meet the following conditions, then the output generated by the List'Output or List'Write subprograms of either instance shall be readable by the List'Input or List'Read of the other instance, respectively:

17/3 - {*AI05-0184-1*} {*AI05-0248-1*} the Element_Type parameters of the two instances are statically matching subtypes of the same type; and

18/3 - {*AI05-0184-1*} the output generated by Element_Type'Output or Element_Type'Write is readable by Element_Type'Input or Element_Type'Read, respectively (where Element_Type denotes the type of the two actual Element_Type parameters).

Implementation Advice

19/3 {*AI05-0001-1*} Bounded list objects should be implemented without implicit pointers or dynamic allocation.

19.a.1/3 **Implementation Advice:** Bounded list objects should be implemented without implicit pointers or dynamic allocation.

20/3 {*AI05-0001-1*} The implementation advice for procedure Move to minimize copying does not apply.

20.a.1/3 **Implementation Advice:** The implementation advice for procedure Move to minimize copying does not apply to bounded lists.

Extensions to Ada 2005

{*AI05-0001-1*} {*AI05-0160-1*} {*AI05-0184-1*} The generic package Containers.Bounded_Doubly_Linked_Lists is new. 20.a/3

Inconsistencies With Ada 2012

{*AI12-0111-1*} **Correction:** Tampering with elements is now defined to be equivalent to tampering with cursors for bounded containers. If a program requires tampering detection to work, it might fail in Ada 2022. Needless to say, this shouldn't happen outside of test programs. See Inconsistencies With Ada 2012 in A.18.2 for more details. 20.b/5

Incompatibilities With Ada 2012

{*AI12-0409-1*} **Correction:**A bounded list now only has Preelaborable_Initialization (abbreviated PI in this note) when the actual for the Element_Type has PI. If an program used a list whose actual Element_Type does not have PI in a context when PI is required (such as a library-level object in a preelaborated unit or as a component of a type with PI), the program would be illegal in Ada 2022 but legal in original Ada 2012. This situation is unlikely, especially as some existing Ada 2012 implementations reject the instance in this case. 20.c/5

A.18.21 The Generic Package Containers.Bounded_Hashed_Maps

{*AI05-0001-1*} The language-defined generic package Containers.Bounded_Hashed_Maps provides a private type Map and a set of operations. It provides the same operations as the package Containers.Hashed_Maps (see A.18.5), with the difference that the maximum storage is bounded. 1/3

Static Semantics

{*AI05-0001-1*} The declaration of the generic library package Containers.Bounded_Hashed_Maps has the same contents and semantics as Containers.Hashed_Maps except: 2/3

- {*AI12-0112-1*} The aspect Preelaborate is replaced with aspect Pure. Aspect Global is deleted. 3/5
- The type Map is declared with discriminants that specify both the capacity (number of elements) and modulus (number of distinct hash values) of the hash table as follows: 4/3

```
{AI12-0112-1}   type Map (Capacity : Count_Type;                    5/5
                Modulus  : Hash_Type) is tagged private...
```

- {*AI12-0409-1*} The aspect_definition for Preelaborable_Initialization for type Map is changed to: 5.1/5

```
Preelaborable_Initialization =>                                     5.2/5
   Element_Type'Preelaborable_Initialization
      and
   Key_Type'Preelaborable_Initialization
```

- The type Map needs finalization if and only if type Key_Type or type Element_Type needs finalization. 6/3

  Implementation Note: {*AI05-0212-1*} The type Map cannot depend on package Ada.Finalization unless the element or key type depends on that package. The objects returned from the Iterator and Reference functions probably do depend on package Ada.Finalization. Restricted environments may need to avoid use of those functions and their associated types. 6.a/3

- {*AI12-0339-1*} In function Empty, the postcondition is altered to: 6.1/5

```
Post =>                                                             6.2/5
   Empty'Result.Capacity = Capacity and then
   Empty'Result.Modulus = Default_Modulus (Capacity) and then
   not Tampering_With_Elements_Prohibited (Empty'Result) and then
   not Tampering_With_Cursors_Prohibited (Empty'Result) and then
   Length (Empty'Result) = 0;
```

- The description of Reserve_Capacity is replaced with: 7/3

```
{AI12-0112-1}    procedure Reserve_Capacity (Container : in out Map;  7.1/5
                                             Capacity  : in     Count_Type)
      with Pre => Capacity <= Container.Capacity
                      or else raise Capacity_Error;
```

8/5 {*AI12-0112-1*} This operation has no effect, [other than checking the precondition].

9/3 • An additional operation is added immediately following Reserve_Capacity:

10/3
```
function Default_Modulus (Capacity : Count_Type) return Hash_Type;
```

11/3 Default_Modulus returns an implementation-defined value for the number of distinct hash values to be used for the given capacity (maximum number of elements).

11.1/5 • {*AI12-0112-1*} For procedures Insert and Include, the part of the precondition reading:

11.2/5
```
(<some length> <= Count_Type'Last - <some other length>
   or else raise Constraint_Error)
```

11.3/5 is replaced by:

11.4/5
```
(<some length> <= Count_Type'Last - <some other length>
   or else raise Constraint_Error) and then
(<some length> > Container.Capacity - <some other length>
   or else raise Capacity_Error)
```

11.5/5 • {*AI12-0112-1*} In procedure Assign, the precondition is altered to:

11.6/5
```
Pre => (not Tampering_With_Cursors_Prohibited (Target)
          or else raise Program_Error) and then
       (Length (Source) <= Target.Capacity
          or else raise Capacity_Error),
```

12/3 • The function Copy is replaced with:

13/5
```
{AI12-0112-1}   function Copy (Source   : Map;
                               Capacity : Count_Type := 0;
                               Modulus  : Hash_Type := 0) return Map
     with Pre  => Capacity = 0 or else Capacity >= Length (Source)
                     or else raise Capacity_Error,
          Post =>
             Length (Copy'Result) = Length (Source) and then
             not Tampering_With_Elements_Prohibited (Copy'Result) and then
             not Tampering_With_Cursors_Prohibited (Copy'Result) and then
             Copy'Result.Capacity = (if Capacity = 0 then
                Length (Source) else Capacity) and then
             Copy'Result.Modulus = (if Modulus = 0 then
                Default_Modulus (Capacity) else Modulus);
```

14/5 {*AI05-0264-1*} Returns a map with key/element pairs initialized from the values in Source.

Bounded (Run-Time) Errors

15/3 {*AI05-0160-1*} {*AI05-0265-1*} It is a bounded error to assign from a bounded map object while tampering with elements [or cursors] of that object is prohibited. Either Program_Error is raised by the assignment, execution proceeds with the target object prohibiting tampering with elements [or cursors], or execution proceeds normally.

15.a/3 **Proof:** Tampering with elements includes tampering with cursors, so we only really need to talk about tampering with elements here; we mention cursors for clarity.

Erroneous Execution

16/3 {*AI05-0265-1*} When a bounded map object *M* is finalized, if tampering with cursors is prohibited for *M* other than due to an assignment from another map, then execution is erroneous.

16.a/3 **Reason:** This is a tampering event, but since the implementation is not allowed to use Ada.Finalization, it is not possible in a pure Ada implementation to detect this error. (There is no Finalize routine that will be called that could make the check.) Since the check probably cannot be made, the bad effects that could occur (such as an iterator going into an infinite loop or accessing a nonexistent element) cannot be prevented and we have to allow anything. We do allow re-assigning an object that only prohibits tampering because it was copied from another object as that cannot cause any negative effects.

Implementation Requirements

{*AI05-0184-1*} {*AI05-0264-1*} For each instance of Containers.Hashed_Maps and each instance of Containers.Bounded_Hashed_Maps, if the two instances meet the following conditions, then the output generated by the Map'Output or Map'Write subprograms of either instance shall be readable by the Map'Input or Map'Read of the other instance, respectively: 17/3

- {*AI05-0184-1*} {*AI05-0248-1*} the Element_Type parameters of the two instances are statically matching subtypes of the same type; and 18/3
- {*AI05-0184-1*} the output generated by Element_Type'Output or Element_Type'Write is readable by Element_Type'Input or Element_Type'Read, respectively (where Element_Type denotes the type of the two actual Element_Type parameters); and 19/3
- {*AI05-0184-1*} the preceding two conditions also hold for the Key_Type parameters of the instances. 20/3

Implementation Advice

{*AI05-0001-1*} {*AI05-0269-1*} Bounded hashed map objects should be implemented without implicit pointers or dynamic allocation. 21/3

Implementation Advice: Bounded hashed map objects should be implemented without implicit pointers or dynamic allocation. 21.a.1/3

{*AI05-0001-1*} The implementation advice for procedure Move to minimize copying does not apply. 22/3

Implementation Advice: The implementation advice for procedure Move to minimize copying does not apply to bounded hashed maps. 22.a.1/3

Extensions to Ada 2005

{*AI05-0001-1*} {*AI05-0160-1*} {*AI05-0184-1*} The generic package Containers.Bounded_Hashed_Maps is new. 22.a/3

Inconsistencies With Ada 2012

{*AI12-0111-1*} **Correction:** Tampering with elements is now defined to be equivalent to tampering with cursors for bounded containers. If a program requires tampering detection to work, it might fail in Ada 2022. Needless to say, this shouldn't happen outside of test programs. See Inconsistencies With Ada 2012 in A.18.2 for more details. 22.b/5

Incompatibilities With Ada 2012

{*AI12-0409-1*} **Correction:**A bounded map now only has Preelaborable_Initialization (abbreviated PI in this note) when the actuals for the Element_Type and the Key_Type have PI. If an program used a map whose actual Element_Type or Key_Type does not have PI in a context when PI is required (such as a library-level object in a preelaborated unit or as a component of a type with PI), the program would be illegal in Ada 2022 but legal in original Ada 2012. This situation is unlikely, especially as some existing Ada 2012 implementations reject the instance in this case. 22.c/5

A.18.22 The Generic Package Containers.Bounded_Ordered_Maps

{*AI05-0001-1*} The language-defined generic package Containers.Bounded_Ordered_Maps provides a private type Map and a set of operations. It provides the same operations as the package Containers.Ordered_Maps (see A.18.6), with the difference that the maximum storage is bounded. 1/3

Static Semantics

{*AI05-0001-1*} The declaration of the generic library package Containers.Bounded_Ordered_Maps has the same contents and semantics as Containers.Ordered_Maps except: 2/3

- {*AI12-0112-1*} The aspect Preelaborate is replaced with aspect Pure. Aspect Global is deleted. 3/5
- The type Map is declared with a discriminant that specifies the capacity (maximum number of elements) as follows: 4/3

5/5 {*AI12-0112-1*} **type** Map (Capacity : Count_Type) **is tagged private**...

5.1/5 • {*AI12-0409-1*} The aspect_definition for Preelaborable_Initialization for type Map is changed to:

5.2/5
```
Preelaborable_Initialization =>
   Element_Type'Preelaborable_Initialization
      and
   Key_Type'Preelaborable_Initialization
```

6/3 • The type Map needs finalization if and only if type Key_Type or type Element_Type needs finalization.

6.a/3 **Implementation Note:** {*AI05-0212-1*} The type Map cannot depend on package Ada.Finalization unless the element type depends on that package. The objects returned from the Iterator and Reference functions probably do depend on package Ada.Finalization. Restricted environments may need to avoid use of those functions and their associated types.

6.1/5 • {*AI12-0339-1*} The function Empty is replaced by:

6.2/5
```
function Empty (Capacity : Count_Type := implementation-defined)
   return Map
   with Post =>
         Empty'Result.Capacity = Capacity and then
         not Tampering_With_Elements_Prohibited (Empty'Result) and then
         not Tampering_With_Cursors_Prohibited (Empty'Result) and then
         Length (Empty'Result) = 0;
```

7/5 • {*AI12-0112-1*} For procedures Insert and Include, the part of the precondition reading:

7.1/5
```
(<some length> <= Count_Type'Last - <some other length>
   or else raise Constraint_Error)
```

7.2/5 is replaced by:

7.3/5
```
(<some length> <= Count_Type'Last - <some other length>
    or else raise Constraint_Error) and then
 (<some length> <= Container.Capacity - <some other length>
    or else raise Capacity_Error)
```

8/5 • {*AI12-0112-1*} In procedure Assign, the precondition is altered to:

8.1/5
```
Pre => (not Tampering_With_Cursors_Prohibited (Target)
          or else raise Program_Error) and then
       (Length (Source) <= Target.Capacity
          or else raise Capacity_Error),
```

9/3 • The function Copy is replaced with:

10/5
```
{AI12-0112-1}   function Copy (Source   : Map;
                  Capacity : Count_Type := 0) return Map
     with Pre  => Capacity = 0 or else Capacity >= Length (Source)
                     or else raise Capacity_Error,
          Post =>
             Length (Copy'Result) = Length (Source) and then
             not Tampering_With_Elements_Prohibited (Copy'Result) and then
             not Tampering_With_Cursors_Prohibited (Copy'Result) and then
             Copy'Result.Capacity = (if Capacity = 0 then
                Length (Source) else Capacity);
```

11/5 {*AI12-0112-1*} Returns a map with key/element pairs initialized from the values in Source.

Bounded (Run-Time) Errors

12/3 {*AI05-0160-1*} {*AI05-0265-1*} It is a bounded error to assign from a bounded map object while tampering with elements [or cursors] of that object is prohibited. Either Program_Error is raised by the assignment, execution proceeds with the target object prohibiting tampering with elements [or cursors], or execution proceeds normally.

12.a/3 **Proof:** Tampering with elements includes tampering with cursors, so we only really need to talk about tampering with elements here; we mention cursors for clarity.

Erroneous Execution

{*AI05-0265-1*} When a bounded map object *M* is finalized, if tampering with cursors is prohibited for *M* other than due to an assignment from another map, then execution is erroneous. 13/3

Reason: This is a tampering event, but since the implementation is not allowed to use Ada.Finalization, it is not possible in a pure Ada implementation to detect this error. (There is no Finalize routine that will be called that could make the check.) Since the check probably cannot be made, the bad effects that could occur (such as an iterator going into an infinite loop or accessing a nonexistent element) cannot be prevented and we have to allow anything. We do allow re-assigning an object that only prohibits tampering because it was copied from another object as that cannot cause any negative effects. 13.a/3

Implementation Requirements

{*AI05-0184-1*} {*AI05-0264-1*} For each instance of Containers.Ordered_Maps and each instance of Containers.Bounded_Ordered_Maps, if the two instances meet the following conditions, then the output generated by the Map'Output or Map'Write subprograms of either instance shall be readable by the Map'Input or Map'Read of the other instance, respectively: 14/3

- {*AI05-0184-1*} {*AI05-0248-1*} the Element_Type parameters of the two instances are statically matching subtypes of the same type; and 15/3
- {*AI05-0184-1*} the output generated by Element_Type'Output or Element_Type'Write is readable by Element_Type'Input or Element_Type'Read, respectively (where Element_Type denotes the type of the two actual Element_Type parameters); and 16/3
- {*AI05-0184-1*} the preceding two conditions also hold for the Key_Type parameters of the instances. 17/3

Implementation Advice

{*AI05-0001-1*} {*AI05-0269-1*} Bounded ordered map objects should be implemented without implicit pointers or dynamic allocation. 18/3

Implementation Advice: Bounded ordered map objects should be implemented without implicit pointers or dynamic allocation. 18.a.1/3

{*AI05-0001-1*} The implementation advice for procedure Move to minimize copying does not apply. 19/3

Implementation Advice: The implementation advice for procedure Move to minimize copying does not apply to bounded ordered maps. 19.a.1/3

Extensions to Ada 2005

{*AI05-0001-1*} {*AI05-0160-1*} {*AI05-0184-1*} The generic package Containers.Bounded_Ordered_Maps is new. 19.a/3

Inconsistencies With Ada 2012

{*AI12-0111-1*} **Correction:** Tampering with elements is now defined to be equivalent to tampering with cursors for bounded containers. If a program requires tampering detection to work, it might fail in Ada 2022. Needless to say, this shouldn't happen outside of test programs. See Inconsistencies With Ada 2012 in A.18.2 for more details. 19.b/5

Incompatibilities With Ada 2012

{*AI12-0409-1*} **Correction:**A bounded map now only has Preelaborable_Initialization (abbreviated PI in this note) when the actuals for the Element_Type and the Key_Type have PI. If an program used a map whose actual Element_Type or Key_Type does not have PI in a context when PI is required (such as a library-level object in a preelaborated unit or as a component of a type with PI), the program would be illegal in Ada 2022 but legal in original Ada 2012. This situation is unlikely, especially as some existing Ada 2012 implementations reject the instance in this case. 19.c/5

A.18.23 The Generic Package Containers.Bounded_Hashed_Sets

1/3 {*AI05-0001-1*} The language-defined generic package Containers.Bounded_Hashed_Sets provides a private type Set and a set of operations. It provides the same operations as the package Containers.Hashed_Sets (see A.18.8), with the difference that the maximum storage is bounded.

Static Semantics

2/3 {*AI05-0001-1*} The declaration of the generic library package Containers.Bounded_Hashed_Sets has the same contents and semantics as Containers.Hashed_Sets except:

3/5 - {*AI12-0112-1*} The aspect Preelaborate is replaced with aspect Pure. Aspect Global is deleted.

4/3 - The type Set is declared with discriminants that specify both the capacity (number of elements) and modulus (number of distinct hash values) of the hash table as follows:

5/5 ```
{AI12-0112-1}   type Set (Capacity : Count_Type;
                          Modulus : Hash_Type) is tagged private...
```

5.1/5 - {*AI12-0409-1*} The aspect_definition for Preelaborable_Initialization for type Set is changed to:

5.2/5 ```
Preelaborable_Initialization =>
   Element_Type'Preelaborable_Initialization
```

6/3 - The type Set needs finalization if and only if type Element_Type needs finalization.

6.a/3 **Implementation Note:** {*AI05-0212-1*} The type Set cannot depend on package Ada.Finalization unless the element or key type depends on that package. The objects returned from the Iterator and Reference functions probably do depend on package Ada.Finalization. Restricted environments may need to avoid use of those functions and their associated types.

6.1/5 - {*AI12-0339-1*} In function Empty, the postcondition is altered to:

6.2/5 ```
Post =>
   Empty'Result.Capacity = Capacity and then
   Empty'Result.Modulus = Default_Modulus (Capacity) and then
   not Tampering_With_Cursors_Prohibited (Empty'Result) and then
   Length (Empty'Result) = 0;
```

7/3 - The description of Reserve_Capacity is replaced with:

7.1/5 ```
{AI12-0112-1}    procedure Reserve_Capacity (Container : in out Set;
                                             Capacity  : in     Count_Type)
      with Pre => Capacity <= Container.Capacity
         or else raise Capacity_Error;
```

8/5 {*AI12-0112-1*} This operation has no effect, [other than checking the precondition].

9/3 - An additional operation is added immediately following Reserve_Capacity:

10/3 ```
function Default_Modulus (Capacity : Count_Type) return Hash_Type;
```

11/3 Default_Modulus returns an implementation-defined value for the number of distinct hash values to be used for the given capacity (maximum number of elements).

11.1/5 - {*AI12-0112-1*} For procedures Insert and Include, the part of the precondition reading:

11.2/5 ```
(<some length> <= Count_Type'Last - <some other length>
   or else raise Constraint_Error)
```

11.3/5 is replaced by:

11.4/5 ```
(<some length> <= Count_Type'Last - <some other length>
   or else raise Constraint_Error) and then
(<some length> <= Container.Capacity - <some other length>
   or else raise Capacity_Error)
```

11.5/5 - {*AI12-0112-1*} In procedure Assign, the precondition is altered to:
```

```
      Pre => (not Tampering_With_Cursors_Prohibited (Target)                    11.6/5
                 or else raise Program_Error) and then
              (Length (Source) <= Target.Capacity
                 or else raise Capacity_Error),
```

- The function Copy is replaced with: 12/3

```
{AI12-0112-1}   function Copy (Source   : Set;                                13/5
                               Capacity : Count_Type := 0;
                               Modulus  : Hash_Type := 0) return Map
     with Pre  => Capacity = 0 or else Capacity >= Length (Source)
                     or else raise Capacity_Error,
          Post =>
             Length (Copy'Result) = Length (Source) and then
             not Tampering_With_Cursors_Prohibited (Copy'Result) and then
             Copy'Result.Capacity = (if Capacity = 0 then
                Length (Source) else Capacity) and then
             Copy'Result.Modulus = (if Modulus = 0 then
                Default_Modulus (Capacity) else Modulus);
```

{*AI05-0264-1*} Returns a set with key/element pairs initialized from the values in Source. 14/5

Bounded (Run-Time) Errors

{*AI05-0160-1*} {*AI05-0265-1*} It is a bounded error to assign from a bounded set object while tampering with elements [or cursors] of that object is prohibited. Either Program_Error is raised by the assignment, execution proceeds with the target object prohibiting tampering with elements [or cursors], or execution proceeds normally. 15/3

Proof: Tampering with elements includes tampering with cursors, so we only really need to talk about tampering with elements here; we mention cursors for clarity. 15.a/3

Erroneous Execution

{*AI05-0265-1*} When a bounded set object *S* is finalized, if tampering with cursors is prohibited for *S* other than due to an assignment from another set, then execution is erroneous. 16/3

Reason: This is a tampering event, but since the implementation is not allowed to use Ada.Finalization, it is not possible in a pure Ada implementation to detect this error. (There is no Finalize routine that will be called that could make the check.) Since the check probably cannot be made, the bad effects that could occur (such as an iterator going into an infinite loop or accessing a nonexistent element) cannot be prevented and we have to allow anything. We do allow re-assigning an object that only prohibits tampering because it was copied from another object as that cannot cause any negative effects. 16.a/3

Implementation Requirements

{*AI05-0184-1*} {*AI05-0264-1*} For each instance of Containers.Hashed_Sets and each instance of Containers.Bounded_Hashed_Sets, if the two instances meet the following conditions, then the output generated by the Set'Output or Set'Write subprograms of either instance shall be readable by the Set'Input or Set'Read of the other instance, respectively: 17/3

- {*AI05-0184-1*} {*AI05-0248-1*} the Element_Type parameters of the two instances are statically matching subtypes of the same type; and 18/3
- {*AI05-0184-1*} the output generated by Element_Type'Output or Element_Type'Write is readable by Element_Type'Input or Element_Type'Read, respectively (where Element_Type denotes the type of the two actual Element_Type parameters). 19/3

Implementation Advice

{*AI05-0001-1*} {*AI05-0269-1*} Bounded hashed set objects should be implemented without implicit pointers or dynamic allocation. 20/3

Implementation Advice: Bounded hashed set objects should be implemented without implicit pointers or dynamic allocation. 20.a.1/3

21/3 {*AI05-0001-1*} The implementation advice for procedure Move to minimize copying does not apply.

21.a.1/3 **Implementation Advice:** The implementation advice for procedure Move to minimize copying does not apply to bounded hashed sets.

Extensions to Ada 2005

21.a/3 {*AI05-0001-1*} {*AI05-0160-1*} {*AI05-0184-1*} The generic package Containers.Bounded_Hashed_Sets is new.

Incompatibilities With Ada 2012

21.b/5 {*AI12-0409-1*} **Correction:**A bounded set now only has Preelaborable_Initialization (abbreviated PI in this note) when the actual for the Element_Type has PI. If an program used a set whose actual Element_Type does not have PI in a context when PI is required (such as a library-level object in a preelaborated unit or as a component of a type with PI), the program would be illegal in Ada 2022 but legal in original Ada 2012. This situation is unlikely, especially as some existing Ada 2012 implementations reject the instance in this case.

A.18.24 The Generic Package Containers.Bounded_Ordered_Sets

1/3 {*AI05-0001-1*} The language-defined generic package Containers.Bounded_Ordered_Sets provides a private type Set and a set of operations. It provides the same operations as the package Containers.Ordered_Sets (see A.18.9), with the difference that the maximum storage is bounded.

Static Semantics

2/3 {*AI05-0001-1*} The declaration of the generic library package Containers.Bounded_Ordered_Sets has the same contents and semantics as Containers.Ordered_Sets except:

3/5 - {*AI12-0112-1*} The aspect Preelaborate is replaced with aspect Pure. Aspect Global is deleted.

4/3 - The type Set is declared with a discriminant that specifies the capacity (maximum number of elements) as follows:

5/5 {*AI12-0112-1*}
```
type Set (Capacity : Count_Type) is tagged private...
```

5.1/5 - {*AI12-0409-1*} The aspect_definition for Preelaborable_Initialization for type Set is changed to:

5.2/5
```
Preelaborable_Initialization =>
   Element_Type'Preelaborable_Initialization
```

6/3 - The type Set needs finalization if and only if type Element_Type needs finalization.

6.a/3 **Implementation Note:** {*AI05-0212-1*} The type Set cannot depend on package Ada.Finalization unless the element type depends on that package. The objects returned from the Iterator and Reference functions probably do depend on package Ada.Finalization. Restricted environments may need to avoid use of those functions and their associated types.

6.1/5 - {*AI12-0339-1*} The function Empty is replaced by:

6.2/5
```
function Empty (Capacity : Count_Type := implementation-defined)
   return Set
   with Post =>
         Empty'Result.Capacity = Capacity and then
         not Tampering_With_Cursors_Prohibited (Empty'Result) and then
         Length (Empty'Result) = 0;
```

7/5 - {*AI12-0112-1*} For procedures Insert and Include, the part of the precondition reading:

7.1/5
```
(<some length> <= Count_Type'Last - <some other length>
   or else raise Constraint_Error)
```

7.2/5 is replaced by:

7.3/5
```
(<some length> <= Count_Type'Last - <some other length>
   or else raise Constraint_Error) and then
(<some length> <= Container.Capacity - <some other length>
   or else raise Capacity_Error)
```

8/5 - {*AI12-0112-1*} In procedure Assign, the precondition is altered to:

```
      Pre => (not Tampering_With_Cursors_Prohibited (Target)                    8.1/5
                or else raise Program_Error) and then
              (Length (Source) <= Target.Capacity
                or else raise Capacity_Error),
```

- The function Copy is replaced with: 9/3

```
{AI12-0112-1}   function Copy (Source   : Set;                                10/5
                  Capacity : Count_Type := 0) return Map
     with Pre  => Capacity = 0 or else Capacity >= Length (Source)
                     or else raise Capacity_Error,
          Post =>
             Length (Copy'Result) = Length (Source) and then
             not Tampering_With_Cursors_Prohibited (Copy'Result) and then
             Copy'Result.Capacity = (if Capacity = 0 then
                Length (Source) else Capacity);
```

{*AI12-0112-1*} Returns a set with key/element pairs initialized from the values in Source. 11/5

Bounded (Run-Time) Errors

{*AI05-0160-1*} {*AI05-0265-1*} It is a bounded error to assign from a bounded set object while tampering with elements [or cursors] of that object is prohibited. Either Program_Error is raised by the assignment, execution proceeds with the target object prohibiting tampering with elements [or cursors], or execution proceeds normally. 12/3

Proof: Tampering with elements includes tampering with cursors, so we only really need to talk about tampering with elements here; we mention cursors for clarity. 12.a/3

Erroneous Execution

{*AI05-0265-1*} When a bounded set object *S* is finalized, if tampering with cursors is prohibited for *S* other than due to an assignment from another set, then execution is erroneous. 13/3

Reason: This is a tampering event, but since the implementation is not allowed to use Ada.Finalization, it is not possible in a pure Ada implementation to detect this error. (There is no Finalize routine that will be called that could make the check.) Since the check probably cannot be made, the bad effects that could occur (such as an iterator going into an infinite loop or accessing a nonexistent element) cannot be prevented and we have to allow anything. We do allow re-assigning an object that only prohibits tampering because it was copied from another object as that cannot cause any negative effects. 13.a/3

Implementation Requirements

{*AI05-0184-1*} {*AI05-0264-1*} For each instance of Containers.Ordered_Sets and each instance of Containers.Bounded_Ordered_Sets, if the two instances meet the following conditions, then the output generated by the Set'Output or Set'Write subprograms of either instance shall be readable by the Set'Input or Set'Read of the other instance, respectively: 14/3

- {*AI05-0184-1*} {*AI05-0248-1*} the Element_Type parameters of the two instances are statically matching subtypes of the same type; and 15/3
- {*AI05-0184-1*} the output generated by Element_Type'Output or Element_Type'Write is readable by Element_Type'Input or Element_Type'Read, respectively (where Element_Type denotes the type of the two actual Element_Type parameters). 16/3

Implementation Advice

{*AI05-0001-1*} {*AI05-0269-1*} Bounded ordered set objects should be implemented without implicit pointers or dynamic allocation. 17/3

Implementation Advice: Bounded ordered set objects should be implemented without implicit pointers or dynamic allocation. 17.a.1/3

{*AI05-0001-1*} The implementation advice for procedure Move to minimize copying does not apply. 18/3

18.a.1/3 **Implementation Advice:** The implementation advice for procedure Move to minimize copying does not apply to bounded ordered sets.

Extensions to Ada 2005

18.a/3 {*AI05-0001-1*} {*AI05-0160-1*} {*AI05-0184-1*} The generic package Containers.Bounded_Ordered_Sets is new.

Incompatibilities With Ada 2012

18.b/5 {*AI12-0409-1*} **Correction:**A bounded set now only has Preelaborable_Initialization (abbreviated PI in this note) when the actual for the Element_Type has PI. If an program used a set whose actual Element_Type does not have PI in a context when PI is required (such as a library-level object in a preelaborated unit or as a component of a type with PI), the program would be illegal in Ada 2022 but legal in original Ada 2012. This situation is unlikely, especially as some existing Ada 2012 implementations reject the instance in this case.

A.18.25 The Generic Package Containers.Bounded_Multiway_Trees

1/3 {*AI05-0136-1*} The language-defined generic package Containers.Bounded_Multiway_Trees provides a private type Tree and a set of operations. It provides the same operations as the package Containers.Multiway_Trees (see A.18.10), with the difference that the maximum storage is bounded.

Static Semantics

2/3 {*AI05-0136-1*} The declaration of the generic library package Containers.Bounded_Multiway_Trees has the same contents and semantics as Containers.Multiway_Trees except:

3/5 - {*AI12-0112-1*} The aspect Preelaborate is replaced with aspect Pure. Aspect Global is deleted.

4/3 - The type Tree is declared with a discriminant that specifies the capacity (maximum number of elements) as follows:

5/5 {*AI12-0112-1*}
```
type Tree (Capacity : Count_Type) is tagged private...
```

5.1/5 - {*AI12-0409-1*} The aspect_definition for Preelaborable_Initialization for type Tree is changed to:

5.2/5
```
Preelaborable_Initialization =>
   Element_Type'Preelaborable_Initialization
```

6/3 - The type Tree needs finalization if and only if type Element_Type needs finalization.

6.a/3 **Implementation Note:** {*AI05-0212-1*} The type Tree cannot depend on package Ada.Finalization unless the element type depends on that package. The objects returned from the Iterator and Reference functions probably do depend on package Ada.Finalization. Restricted environments may need to avoid use of those functions and their associated types.

6.1/5 - {*AI12-0339-1*} The function Empty is replaced by:

6.2/5
```
function Empty (Capacity : Count_Type := implementation-defined)
   return Tree
   with Post =>
         Empty'Result.Capacity = Capacity and then
         not Tampering_With_Elements_Prohibited (Empty'Result) and then
         not Tampering_With_Cursors_Prohibited (Empty'Result) and then
         Node_Count (Empty'Result) = 1;
```

7/5 - {*AI12-0112-1*} For procedures Insert_Child, Prepend_Child, and Append_Child, the initial subexpression of the precondition is replaced with:

7.1/5
```
with Pre => (not Tampering_With_Cursors_Prohibited (Container)
                or else raise Program_Error) and then
             (Node_Count (Container) - 1 <= Container.Capacity - Count
                or else raise Capacity_Error)
```

8/5 - {*AI12-0112-1*} In procedure Assign, the precondition is altered to:

8.1/5
```
Pre => (not Tampering_With_Cursors_Prohibited (Target)
           or else raise Program_Error) and then
        (Node_Count (Source) - 1 <= Target.Capacity
           or else raise Capacity_Error),
```

- Function Copy is declared as follows: 9/3

{*AI12-0056-1*} 10/4
```
function Copy (Source : Tree; Capacity : Count_Type := 0)
   return Tree
   with Pre  => Capacity = 0 or else Capacity >= Node_Count (Source) - 1
                   or else raise Capacity_Error,
        Post =>
           Node_Count (Copy'Result) = Node_Count (Source) and then
           not Tampering_With_Elements_Prohibited (Copy'Result) and then
           not Tampering_With_Cursors_Prohibited (Copy'Result) and then
           Copy'Result.Capacity = (if Capacity = 0 then
              Node_Count (Source) - 1 else Capacity);
```

Returns a list whose elements have the same values as the elements of Source. 11/5

- {*AI12-0112-1*} In the four-parameter procedure Copy_Subtree, the last **or else** of the precondition is replaced by: 11.1/5

```
(not Is_Root (Source)                                                    11.2/5
   or else raise Constraint_Error) and then
(Node_Count (Target) - 1 + Subtree_Node_Count (Source) <=
   Target.Capacity
   or else raise Capacity_Error),
```

- {*AI12-0112-1*} In the five-parameter procedure Copy_Subtree, the last **or else** of the precondition is replaced by: 11.3/5

```
(not Is_Root (Source, Subtree)                                           11.4/5
   or else raise Constraint_Error) and then
(Node_Count (Target) - 1 +
      Subtree_Node_Count (Source, Subtree) <= Target.Capacity
   or else raise Capacity_Error),
```

- {*AI12-0112-1*} In Copy_Local_Subtree, the last **or else** of the precondition is replaced by: 11.5/5

```
(not Is_Root (Source, Subtree)                                           11.6/5
   or else raise Constraint_Error) and then
(Node_Count (Target) - 1 +
      Subtree_Node_Count (Target, Source) <= Target.Capacity
   or else raise Capacity_Error),
```

- {*AI05-0136-1*} {*AI05-0248-1*} {*AI12-0112-1*} In the five-parameter procedure Splice_Subtree, the penultimate **or else** of the precondition is replaced by: 12/5

```
(Has_Element (Source, Position)                                          12.1/5
   or else raise Program_Error) and then
(Target'Has_Same_Storage (Source) or else
 Node_Count (Target) - 1 +
   Subtree_Node_Count (Source, Position) <= Target.Capacity
   or else raise Capacity_Error) and then
```

- {*AI05-0136-1*} {*AI05-0248-1*} {*AI12-0112-1*} In the five-parameter procedure Splice_Children, the penultimate **elsif** of the precondition is replaced by: 13/5

```
(Before = No_Element or else                                             13.1/5
 Parent (Target, Before) /= Target_Parent
   or else raise Constraint_Error) and then
(Target'Has_Same_Storage (Source) or else
 Node_Count (Target) - 1 +
   Child_Count (Source, Source_Parent) <= Target.Capacity
   or else raise Capacity_Error) and then
```

Bounded (Run-Time) Errors

{*AI05-0160-1*} {*AI05-0265-1*} It is a bounded error to assign from a bounded tree object while tampering with elements [or cursors] of that object is prohibited. Either Program_Error is raised by the assignment, execution proceeds with the target object prohibiting tampering with elements [or cursors], or execution proceeds normally. 14/3

14.a/3 **Proof:** Tampering with elements includes tampering with cursors, so we only really need to talk about tampering with elements here; we mention cursors for clarity.

Erroneous Execution

15/3 {*AI05-0265-1*} When a bounded tree object *T* is finalized, if tampering with cursors is prohibited for *T* other than due to an assignment from another tree, then execution is erroneous.

15.a/3 **Reason:** This is a tampering event, but since the implementation is not allowed to use Ada.Finalization, it is not possible in a pure Ada implementation to detect this error. (There is no Finalize routine that will be called that could make the check.) Since the check probably cannot be made, the bad effects that could occur (such as an iterator going into an infinite loop or accessing a nonexistent element) cannot be prevented and we have to allow anything. We do allow re-assigning an object that only prohibits tampering because it was copied from another object as that cannot cause any negative effects.

Implementation Requirements

16/3 {*AI05-0184-1*} {*AI05-0264-1*} For each instance of Containers.Multiway_Trees and each instance of Containers.Bounded_Multiway_Trees, if the two instances meet the following conditions, then the output generated by the Tree'Output or Tree'Write subprograms of either instance shall be readable by the Tree'Input or Tree'Read of the other instance, respectively:

17/3 - {*AI05-0184-1*} {*AI05-0248-1*} the Element_Type parameters of the two instances are statically matching subtypes of the same type; and

18/3 - {*AI05-0184-1*} the output generated by Element_Type'Output or Element_Type'Write is readable by Element_Type'Input or Element_Type'Read, respectively (where Element_Type denotes the type of the two actual Element_Type parameters).

Implementation Advice

19/3 {*AI05-0136-1*} Bounded tree objects should be implemented without implicit pointers or dynamic allocation.

19.a.1/3 **Implementation Advice:** Bounded tree objects should be implemented without implicit pointers or dynamic allocation.

20/3 {*AI05-0136-1*} The implementation advice for procedure Move to minimize copying does not apply.

20.a.1/3 **Implementation Advice:** The implementation advice for procedure Move to minimize copying does not apply to bounded trees.

Extensions to Ada 2005

20.a/3 {*AI05-0136-1*} {*AI05-0184-1*} The generic package Containers.Bounded_Multiway_Trees is new.

Inconsistencies With Ada 2012

20.b/5 {*AI12-0111-1*} **Correction:** Tampering with elements is now defined to be equivalent to tampering with cursors for bounded containers. If a program requires tampering detection to work, it might fail in Ada 2022. Needless to say, this shouldn't happen outside of test programs. See Inconsistencies With Ada 2012 in A.18.2 for more details.

Incompatibilities With Ada 2012

20.c/5 {*AI12-0409-1*} **Correction:**A bounded tree now only has Preelaborable_Initialization (abbreviated PI in this note) when the actual for the Element_Type has PI. If an program used a tree whose actual Element_Type does not have PI in a context when PI is required (such as a library-level object in a preelaborated unit or as a component of a type with PI), the program would be illegal in Ada 2022 but legal in original Ada 2012. This situation is unlikely, especially as some existing Ada 2012 implementations reject the instance in this case.

A.18.26 Array Sorting

{*AI95-00302-03*} {*AI05-0001-1*} The language-defined generic procedures Containers.Generic_-Array_Sort, Containers.Generic_Constrained_Array_Sort, and Containers.Generic_Sort provide sorting on arbitrary array types. 1/3

Static Semantics

{*AI95-00302-03*} The generic library procedure Containers.Generic_Array_Sort has the following declaration: 2/2

```
{AI12-0112-1} generic                                                    3/5
   type Index_Type is (<>);
   type Element_Type is private;
   type Array_Type is array (Index_Type range <>) of Element_Type;
   with function "<" (Left, Right : Element_Type)
      return Boolean is <>;
procedure Ada.Containers.Generic_Array_Sort (Container : in out Array_Type)
   with Pure, Nonblocking, Global => null;
```

Discussion: {*AI12-0112-1*} Global => **null** means that the only global side-effects allowed are associated with the actual generic parameters. Similarly, when Nonblocking is set to True for a generic unit, the only blocking allowed is that associated with the actual generic parameters. 3.a/5

Reorders the elements of Container such that the elements are sorted smallest first as determined by the generic formal "<" operator provided. Any exception raised during evaluation of "<" is propagated. 4/2

{*AI05-0044-1*} {*AI05-0262-1*} The actual function for the generic formal function "<" of Generic_Array_Sort is expected to return the same value each time it is called with a particular pair of element values. It should define a strict weak ordering relationship (see A.18); it should not modify Container. If the actual for "<" behaves in some other manner, the behavior of the instance of Generic_Array_Sort is unspecified. The number of times Generic_Array_Sort calls "<" is unspecified. 5/3

Ramification: This implies swapping the elements, usually including an intermediate copy. This of course means that the elements will be copied. Since the elements are nonlimited, this usually will not be a problem. Note that there is Implementation Advice below that the implementation should use a sort that minimizes copying of elements. 5.a/2

The sort is not required to be stable (and the fast algorithm required will not be stable). If a stable sort is needed, the user can include the original location of the element as an extra "sort key". We considered requiring the implementation to do that, but it is mostly extra overhead -- usually there is something already in the element that provides the needed stability. 5.b/2

{*AI95-00302-03*} The generic library procedure Containers.Generic_Constrained_Array_Sort has the following declaration: 6/2

```
{AI12-0112-1} generic                                                    7/5
   type Index_Type is (<>);
   type Element_Type is private;
   type Array_Type is array (Index_Type) of Element_Type;
   with function "<" (Left, Right : Element_Type)
      return Boolean is <>;
procedure Ada.Containers.Generic_Constrained_Array_Sort
      (Container : in out Array_Type)
   with Pure, Nonblocking, Global => null;
```

Discussion: {*AI12-0112-1*} Global => **null** means that the only global side-effects allowed are associated with the actual generic parameters. Similarly, when Nonblocking is set to True for a generic unit, the only blocking allowed is that associated with the actual generic parameters. 7.a/5

Reorders the elements of Container such that the elements are sorted smallest first as determined by the generic formal "<" operator provided. Any exception raised during evaluation of "<" is propagated. 8/2

9/3 {*AI05-0044-1*} {*AI05-0262-1*} The actual function for the generic formal function "<" of Generic_Constrained_Array_Sort is expected to return the same value each time it is called with a particular pair of element values. It should define a strict weak ordering relationship (see A.18); it should not modify Container. If the actual for "<" behaves in some other manner, the behavior of the instance of Generic_Constrained_Array_Sort is unspecified. The number of times Generic_Constrained_Array_Sort calls "<" is unspecified.

9.1/3 {*AI05-0001-1*} The generic library procedure Containers.Generic_Sort has the following declaration:

9.2/5
```
{AI12-0056-1} {AI12-0112-1} generic
   type Index_Type is (<>);
   with function Before (Left, Right : Index_Type) return Boolean;
   with procedure Swap (Left, Right : in Index_Type);
procedure Ada.Containers.Generic_Sort
      (First, Last : Index_Type'Base)
   with Pure, Nonblocking, Global => null;
```

9.a/5 **Discussion:** {*AI12-0112-1*} Global => **null** means that the only global side-effects allowed are associated with the actual generic parameters. Similarly, when Nonblocking is set to True for a generic unit, the only blocking allowed is that associated with the actual generic parameters.

9.3/3 {*AI05-0001-1*} {*AI05-0248-1*} Reorders the elements of an indexable structure, over the range First .. Last, such that the elements are sorted in the ordering determined by the generic formal function Before; Before should return True if Left is to be sorted before Right. The generic formal Before compares the elements having the given indices, and the generic formal Swap exchanges the values of the indicated elements. Any exception raised during evaluation of Before or Swap is propagated.

9.4/3 The actual function for the generic formal function Before of Generic_Sort is expected to return the same value each time it is called with index values that identify a particular pair of element values. It should define a strict weak ordering relationship (see A.18); it should not modify the elements. The actual function for the generic formal Swap should exchange the values of the indicated elements. If the actual for either Before or Swap behaves in some other manner, the behavior of Generic_Sort is unspecified. The number of times the Generic_Sort calls Before or Swap is unspecified.

Implementation Advice

10/2 {*AI95-00302-03*} The worst-case time complexity of a call on an instance of Containers.Generic_Array_Sort or Containers.Generic_Constrained_Array_Sort should be $O(N^{**}2)$ or better, and the average time complexity should be better than $O(N^{**}2)$, where N is the length of the Container parameter.

10.a/2 **Implementation Advice:** Containers.Generic_Array_Sort and Containers.Generic_Constrained_Array_Sort should have an average time complexity better than $O(N^{**}2)$ and worst case no worse than $O(N^{**}2)$.

10.b/2 **Discussion:** In other words, we're requiring the use of a sorting algorithm better than $O(N^{**}2)$, such as Quicksort. No bubble sorts allowed!

11/2 {*AI95-00302-03*} Containers.Generic_Array_Sort and Containers.Generic_Constrained_Array_Sort should minimize copying of elements.

11.a/2 **Implementation Advice:** Containers.Generic_Array_Sort and Containers.Generic_Constrained_Array_Sort should minimize copying of elements.

11.b/2 **To be honest:** We do not mean “absolutely minimize” here; we're not intending to require a single copy for each element. Rather, we want to suggest that the sorting algorithm chosen is one that does not copy items unnecessarily. Bubble sort would not meet this advice, for instance.

12/3 {*AI05-0248-1*} The worst-case time complexity of a call on an instance of Containers.Generic_Sort should be $O(N^{**}2)$ or better, and the average time complexity should be better than $O(N^{**}2)$, where N is the difference between the Last and First parameters plus 1.

Implementation Advice: Containers.Generic_Sort should have an average time complexity better than *O*(*N***2) and worst case no worse than *O*(*N***2). 12.a.1/3

{*AI05-0248-1*} Containers.Generic_Sort should minimize calls to the generic formal Swap. 13/3

Implementation Advice: Containers.Generic_Sort should minimize calls to the generic formal Swap. 13.a.1/3

Extensions to Ada 95

{*AI95-00302-03*} The generic procedures Containers.Generic_Array_Sort and Containers.Generic_Constrained_Array_Sort are new. 13.a/2

Extensions to Ada 2005

{*AI05-0001-1*} {*AI05-0248-1*} The generic procedure Containers.Generic_Sort is new. 13.b/3

Wording Changes from Ada 2005

{*AI05-0044-1*} **Correction:** Redefined "<" actuals to require a strict weak ordering; the old definition allowed indeterminant comparisons that would not have worked in a sort. 13.c/3

Wording Changes from Ada 2012

{*AI12-0112-1*} Added contracts to these procedures. 13.d/5

A.18.27 The Generic Package Containers.Synchronized_Queue_Interfaces

{*AI05-0159-1*} The language-defined generic package Containers.Synchronized_Queue_Interfaces provides interface type Queue, and a set of operations for that type. Interface Queue specifies a first-in, first-out queue. 1/3

Static Semantics

{*AI05-0159-1*} The generic library package Containers.Synchronized_Queue_Interfaces has the following declaration: 2/3

```
{AI12-0112-1} generic                                                                    3/5
   type Element_Type is private;
package Ada.Containers.Synchronized_Queue_Interfaces
   with Pure, Nonblocking, Global => null is
```

Discussion: {*AI12-0112-1*} Global => **null** means that the only global side-effects allowed are associated with the actual generic parameters. Similarly, when Nonblocking is set to True for a generic unit, the only blocking allowed is that associated with the actual generic parameters. 3.a/5

```
   type Queue is synchronized interface;                                                 4/3

{AI12-0112-1}    procedure Enqueue                                                       5/5
     (Container : in out Queue;
      New_Item  : in     Element_Type) is abstract
       with Synchronization => By_Entry,
            Nonblocking => False,
            Global'Class=> in out synchronized;

{AI12-0112-1}    procedure Dequeue                                                       6/5
     (Container : in out Queue;
      Element   :    out Element_Type) is abstract
       with Synchronization => By_Entry,
            Nonblocking => False,
            Global'Class=> in out synchronized;

{AI12-0112-1}    function Current_Use (Container : Queue) return Count_Type is          7/5
abstract
       with Nonblocking, Global'Class => null, Use_Formal => null;
   function Peak_Use (Container : Queue) return Count_Type is abstract
       with Nonblocking, Global'Class => null, Use_Formal => null,
            Post'Class => Peak_Use'Result >= Current_Use (Container);
```

8/3
```
end Ada.Containers.Synchronized_Queue_Interfaces;
```

8.1/5 {*AI12-0112-1*} The subprogram behavior descriptions given below are the semantics for the corresponding callable entities found in the language-defined generic packages that have a formal package named Queue_Interfaces.

9/5
```
{AI12-0112-1} procedure Enqueue
  (Container : in out Queue;
   New_Item  : in     Element_Type) is abstract
   with Synchronization => By_Entry
        Nonblocking => False,
        Global'Class=> in out synchronized;
```

10/3 {*AI05-0159-1*} {*AI05-0262-1*} {*AI05-0264-1*} A queue type that implements this interface is allowed to have a bounded *capacity*. If the queue object has a bounded capacity, and the number of existing elements equals the capacity, then Enqueue blocks until storage becomes available; otherwise, Enqueue does not block. In any case, it then copies New_Item onto the queue.

11/5
```
{AI12-0112-1} procedure Dequeue
  (Container : in out Queue;
   Element   :    out Element_Type) is abstract
   with Synchronization => By_Entry
        Nonblocking => False,
        Global'Class=> in out synchronized;
```

12/3 {*AI05-0159-1*} {*AI05-0251-1*} If the queue is empty, then Dequeue blocks until an item becomes available. In any case, it then assigns the element at the head of the queue to Element, and removes it from the queue.

13/5
```
{AI12-0112-1} function Current_Use (Container : Queue) return Count_Type is
abstract
   with Nonblocking, Global'Class=> null, Use_Formal => null;
```

14/3 {*AI05-0159-1*} Returns the number of elements currently in the queue.

15/5
```
{AI12-0112-1} function Peak_Use (Container : Queue) return Count_Type is abstract
   with Nonblocking, Global'Class=> null, Use_Formal => null,
         Post'Class => Peak_Use'Result >= Current_Use (Container);
```

16/3 {*AI05-0159-1*} Returns the maximum number of elements that have been in the queue at any one time.

17/3 NOTE {*AI05-0251-1*} Unlike other language-defined containers, there are no queues whose element types are indefinite. Elements of an indefinite type can be handled by defining the element of the queue to be a holder container (see A.18.18) of the indefinite type, or to be an explicit access type that designates the indefinite type.

17.a/3 **Reason:** There are no indefinite queues, as a useful definition for Dequeue is not possible. Dequeue cannot be a function, as Ada does not have entries that are functions (thus conditional and timed calls would not be possible). Moreover, protected functions do not allow modifying the queue object (thus it doesn't work even if we decided we didn't care about conditional and timed calls). If Dequeue is an entry, then the dequeued object would have to be an **out** parameter and that would require the queue client to guess the tag and constraints of the value that will be dequeued (otherwise Constraint_Error would be raised), and that is rarely going to be possible.

Extensions to Ada 2005

17.b/3 {*AI05-0159-1*} {*AI05-0251-1*} The generic package Containers.Synchronized_Queue_Interfaces is new.

A.18.28 The Generic Package Containers.Unbounded_Synchronized_Queues

Static Semantics

{*AI05-0159-1*} The language-defined generic package Containers.Unbounded_Synchronized_Queues provides type Queue, which implements the interface type Containers.Synchronized_Queue_Interfaces.Queue. 1/3

```
{AI12-0112-1} with System;                                                    2/5
with Ada.Containers.Synchronized_Queue_Interfaces;
generic
   with package Queue_Interfaces is
      new Ada.Containers.Synchronized_Queue_Interfaces (<>);
   Default_Ceiling : System.Any_Priority := System.Priority'Last;
package Ada.Containers.Unbounded_Synchronized_Queues
   with Preelaborate,
        Nonblocking, Global => in out synchronized is
```

Discussion: {*AI12-0112-1*} For discussion on the reasons and meaning of the specifications of the Global and Nonblocking aspects of this generic package, see the notes on the specification of the Containers.Vectors package (see A.18.2). 2.a/5

```
   package Implementation is                                                  3/3
      ... -- not specified by the language
   end Implementation;

   protected type Queue                                                       4/3
        (Ceiling : System.Any_Priority := Default_Ceiling)
           with Priority => Ceiling is
        new Queue_Interfaces.Queue with

      overriding                                                              5/3
      entry Enqueue (New_Item : in Queue_Interfaces.Element_Type);
      overriding
      entry Dequeue (Element : out Queue_Interfaces.Element_Type);

{AI12-0112-1}       overriding                                              6/5
      function Current_Use return Count_Type
         with Nonblocking, Global => null, Use_Formal => null;
      overriding
      function Peak_Use return Count_Type
         with Nonblocking, Global => null, Use_Formal => null;

   private                                                                    7/3
      ... -- not specified by the language
   end Queue;

private                                                                       8/3

   ... -- not specified by the language                                       9/3

end Ada.Containers.Unbounded_Synchronized_Queues;                             10/3
```

{*AI05-0159-1*} The type Queue is used to represent task-safe queues. 11/3

{*AI05-0159-1*} The capacity for instances of type Queue is unbounded. 12/3

Ramification: Enqueue never blocks; if more storage is needed for a new element, it is allocated dynamically. We don't need to explicitly specify that Queue needs finalization, because it is visibly protected. 12.a/3

Discussion: Nested package Implementation can be used to declare the types needed to implement the protected type Queue. This nested package is necessary as types cannot be declared in the private part of a protected type, and the types have to be declared within the generic unit in order to depend on the types imported with package Queue_Interfaces. Clients should never depend on the contents of nested package Implementation. 12.b/3

Extensions to Ada 2005

{*AI05-0159-1*} The generic package Containers.Unbounded_Synchronized_Queues is new. 12.c/3

A.18.29 The Generic Package Containers.Bounded_Synchronized_Queues

Static Semantics

1/3 {*AI05-0159-1*} The language-defined generic package Containers.Bounded_Synchronized_Queues provides type Queue, which implements the interface type Containers.Synchronized_Queue_Interfaces.Queue.

2/5
```
{AI12-0112-1} with System;
with Ada.Containers.Synchronized_Queue_Interfaces;
generic
   with package Queue_Interfaces is
      new Ada.Containers.Synchronized_Queue_Interfaces (<>);
   Default_Capacity : Count_Type;
   Default_Ceiling  : System.Any_Priority := System.Priority'Last;
package Ada.Containers.Bounded_Synchronized_Queues
   with Preelaborate,
        Nonblocking, Global => in out synchronized is
```

2.a/5 **Discussion:** {*AI12-0112-1*} For discussion on the reasons and meaning of the specifications of the Global and Nonblocking aspects of this generic package, see the notes on the specification of the Containers.Vectors package (see A.18.2).

3/3
```
   package Implementation is
      ... -- not specified by the language
   end Implementation;
```

4/3
```
   protected type Queue
        (Capacity : Count_Type := Default_Capacity;
         Ceiling  : System.Any_Priority := Default_Ceiling)
           with Priority => Ceiling is
        new Queue_Interfaces.Queue with
```

5/3
```
      overriding
      entry Enqueue (New_Item : in Queue_Interfaces.Element_Type);
      overriding
      entry Dequeue (Element : out Queue_Interfaces.Element_Type);
```

6/5
```
   {AI12-0112-1}      overriding
      function Current_Use return Count_Type
         with Nonblocking, Global => null, Use_Formal => null;
      overriding
      function Peak_Use return Count_Type
         with Nonblocking, Global => null, Use_Formal => null;
```

7/3
```
   private
      ... -- not specified by the language
   end Queue;
```

8/3
```
private
```

9/3
```
   ... -- not specified by the language
```

10/3
```
end Ada.Containers.Bounded_Synchronized_Queues;
```

11/3 {*AI05-0159-1*} The semantics are the same as for Unbounded_Synchronized_Queues, except:

12/3 - The capacity for instances of type Queue is bounded and specified by the discriminant Capacity.

12.a/3 **Ramification:** Since this type has a bounded capacity, Enqueue might block if the queue is full.

Implementation Advice

13/3 {*AI05-0159-1*} Bounded queue objects should be implemented without implicit pointers or dynamic allocation.

13.a.1/3 **Implementation Advice:** Bounded queue objects should be implemented without implicit pointers or dynamic allocation.

Extensions to Ada 2005

{*AI05-0159-1*} The generic package Containers.Bounded_Synchronized_Queues is new. 13.a/3

A.18.30 The Generic Package Containers.Unbounded_Priority_Queues

Static Semantics

{*AI05-0159-1*} The language-defined generic package Containers.Unbounded_Priority_Queues provides type Queue, which implements the interface type Containers.Synchronized_Queue_Interfaces.Queue. 1/3

```
{AI12-0112-1} with System;                                                    2/5
with Ada.Containers.Synchronized_Queue_Interfaces;
generic
   with package Queue_Interfaces is
      new Ada.Containers.Synchronized_Queue_Interfaces (<>);
   type Queue_Priority is private;
   with function Get_Priority
     (Element : Queue_Interfaces.Element_Type) return Queue_Priority is <>;
   with function Before
     (Left, Right : Queue_Priority) return Boolean is <>;
   Default_Ceiling : System.Any_Priority := System.Priority'Last;
package Ada.Containers.Unbounded_Priority_Queues
   with Preelaborate,
        Nonblocking, Global => in out synchronized is
```

Discussion: {*AI12-0112-1*} For discussion on the reasons and meaning of the specifications of the Global and Nonblocking aspects of this generic package, see the notes on the specification of the Containers.Vectors package (see A.18.2). 2.a/5

```
   package Implementation is                                                  3/3
      ... -- not specified by the language
   end Implementation;

   protected type Queue                                                       4/3
        (Ceiling : System.Any_Priority := Default_Ceiling)
           with Priority => Ceiling is
        new Queue_Interfaces.Queue with

      overriding                                                              5/3
      entry Enqueue (New_Item : in Queue_Interfaces.Element_Type);
      overriding
      entry Dequeue (Element : out Queue_Interfaces.Element_Type);

{AI05-0159-1} {AI05-0251-1}       not overriding                            6/3
      procedure Dequeue_Only_High_Priority
        (At_Least : in     Queue_Priority;
         Element  : in out Queue_Interfaces.Element_Type;
         Success  :    out Boolean);

{AI12-0112-1}       overriding                                              7/5
      function Current_Use return Count_Type
         with Nonblocking, Global => null, Use_Formal => null;
      overriding
      function Peak_Use return Count_Type
         with Nonblocking, Global => null, Use_Formal => null;

   private                                                                    8/3
      ... -- not specified by the language
   end Queue;

private                                                                       9/3

   ... -- not specified by the language                                       10/3

end Ada.Containers.Unbounded_Priority_Queues;                                 11/3
```

{*AI05-0159-1*} The type Queue is used to represent task-safe priority queues. 12/3

{*AI05-0159-1*} The capacity for instances of type Queue is unbounded. 13/3

14/3 {*AI05-0159-1*} Two elements *E1* and *E2* are equivalent if Before(Get_Priority(*E1*), Get_Priority(*E2*)) and Before(Get_Priority(*E2*), Get_Priority(*E1*)) both return False.

15/3 {*AI05-0159-1*} {*AI05-0248-1*} The actual functions for Get_Priority and Before are expected to return the same value each time they are called with the same actuals, and should not modify their actuals. Before should define a strict weak ordering relationship (see A.18). If the actual functions behave in some other manner, the behavior of Unbounded_Priority_Queues is unspecified.

16/3 {*AI05-0159-1*} Enqueue inserts an item according to the order specified by the Before function on the result of Get_Priority on the elements; Before should return True if Left is to be inserted before Right. If the queue already contains elements equivalent to New_Item, then it is inserted after the existing equivalent elements.

16.a/3 **Ramification:** Enqueue never blocks; if more storage is needed for a new element, it is allocated dynamically. We don't need to explicitly specify that Queue needs finalization, because it is visibly protected.

17/3 {*AI05-0159-1*} {*AI05-0251-1*} {*AI05-0262-1*} For a call on Dequeue_Only_High_Priority, if the head of the nonempty queue is *E*, and the function Before(At_Least, Get_Priority(*E*)) returns False, then *E* is assigned to Element and then removed from the queue, and Success is set to True; otherwise, Success is set to False and Element is unchanged.

17.a/3 **Ramification:** {*AI05-0251-1*} Unlike Dequeue, Dequeue_Only_High_Priority is not blocking; it always returns immediately.

17.b/3 **Reason:** {*AI05-0251-1*} The use of Before is "backwards" so that it acts like ">=" (it is defined similarly to ">"); thus we dequeue only when it is False.

Extensions to Ada 2005

17.c/3 {*AI05-0159-1*} {*AI05-0251-1*} The generic package Containers.Unbounded_Priority_Queues is new.

A.18.31 The Generic Package Containers.Bounded_Priority_Queues

Static Semantics

1/3 {*AI05-0159-1*} The language-defined generic package Containers.Bounded_Priority_Queues provides type Queue, which implements the interface type Containers.Synchronized_Queue_Interfaces.Queue.

2/5
```
{AI12-0112-1} with System;
with Ada.Containers.Synchronized_Queue_Interfaces;
generic
   with package Queue_Interfaces is
      new Ada.Containers.Synchronized_Queue_Interfaces (<>);
   type Queue_Priority is private;
   with function Get_Priority
     (Element : Queue_Interfaces.Element_Type) return Queue_Priority is <>;
   with function Before
     (Left, Right : Queue_Priority) return Boolean is <>;
   Default_Capacity : Count_Type;
   Default_Ceiling  : System.Any_Priority := System.Priority'Last;
package Ada.Containers.Bounded_Priority_Queues
   with Preelaborate,
        Nonblocking, Global => in out synchronized is
```

2.a/5 **Discussion:** {*AI12-0112-1*} For discussion on the reasons and meaning of the specifications of the Global and Nonblocking aspects of this generic package, see the notes on the specification of the Containers.Vectors package (see A.18.2).

3/3
```
   package Implementation is
      ... -- not specified by the language
   end Implementation;
```

```
   protected type Queue                                                    4/3
        (Capacity : Count_Type := Default_Capacity;
         Ceiling  : System.Any_Priority := Default_Ceiling)
           with Priority => Ceiling is
      new Queue_Interfaces.Queue with

      overriding                                                           5/3
      entry Enqueue (New_Item : in Queue_Interfaces.Element_Type);
      overriding
      entry Dequeue (Element : out Queue_Interfaces.Element_Type);
{AI05-0159-1} {AI05-0251-1}       not overriding                         6/3
      procedure Dequeue_Only_High_Priority
        (At_Least : in     Queue_Priority;
         Element  : in out Queue_Interfaces.Element_Type;
         Success  :    out Boolean);
{AI12-0112-1}       overriding                                           7/5
      function Current_Use return Count_Type
         with Nonblocking, Global => null, Use_Formal => null;
      overriding
      function Peak_Use return Count_Type
         with Nonblocking, Global => null, Use_Formal => null;

   private                                                                 8/3
      ... -- not specified by the language
   end Queue;

private                                                                   9/3

   ... -- not specified by the language                                   10/3

end Ada.Containers.Bounded_Priority_Queues;                               11/3
```

{*AI05-0159-1*} The semantics are the same as for Unbounded_Priority_Queues, except: 12/3

- The capacity for instances of type Queue is bounded and specified by the discriminant Capacity. 13/3

  Ramification: Since this type has a bounded capacity, Enqueue might block if the queue is full. 13.a/3

Implementation Advice

{*AI05-0159-1*} Bounded priority queue objects should be implemented without implicit pointers or dynamic allocation. 14/3

Implementation Advice: Bounded priority queue objects should be implemented without implicit pointers or dynamic allocation. 14.a.1/3

Extensions to Ada 2005

{*AI05-0159-1*} {*AI05-0251-1*} The generic package Containers.Bounded_Priority_Queues is new. 14.a/3

A.18.32 The Generic Package Containers.Bounded_Indefinite_Holders

{*AI12-0254-1*} The language-defined generic package Containers.Bounded_Indefinite_Holders provides a private type Holder and a set of operations for that type. It provides the same operations as the package Containers.Indefinite_Holders (see A.18.18), with the difference that the maximum storage is bounded. 1/5

Static Semantics

{*AI12-0254-1*} The declaration of the generic library package Containers.Bounded_Indefinite_Holders has the same contents and semantics as Containers.Indefinite_Holders except: 2/5

- The following is added to the context clause: 3/5

```
with System.Storage_Elements; use System.Storage_Elements;                4/5
```

- An additional generic parameter follows Element_Type: 5/5

```
Max_Element_Size_in_Storage_Elements : Storage_Count;                      6/5
```

6.a/5 **Reason:** This value is in Storage_Elements so that it can be used as a discriminant on a storage pool object in the implementation of the object; Ada doesn't allow discriminant dependent components to use formulas.

6.b/5 This is a generic parameter as it is a property of the Element_Type; the largest possible object of Element_Type is unlikely to be different for different containers, so making it a discriminant (as Capacity is) provides no useful capability.

7/5 - {*AI12-0409-1*} The aspect_definition for Preelaborable_Initialization for type Holder is changed to:

8/5
```
Preelaborable_Initialization =>
   Element_Type'Preelaborable_Initialization
```

9/5 - Add to the precondition of To_Holder and Replace_Element:

10/5
```
and then (New_Item'Size <=
             Max_Element_Size_in_Storage_Elements * System.Storage_Unit
          or else raise Program_Error)
```

10.a/5 **Reason:** This ensures that an object that won't fit is not inserted into the container.

Bounded (Run-Time) Errors

11/5 {*AI12-0254-1*} It is a bounded error to assign from a bounded holder object while tampering with elements of that object is prohibited. Either Program_Error is raised by the assignment, execution proceeds with the target object prohibiting tampering with elements, or execution proceeds normally.

Implementation Requirements

12/5 {*AI12-0254-1*} For each instance of Containers.Indefinite_Holders and each instance of Containers.Bounded_Indefinite_Holders, if the two instances meet the following conditions, then the output generated by the Holder'Output or Holder'Write subprograms of either instance shall be readable by the Holder'Input or Holder'Read of the other instance, respectively:

13/5 - the Element_Type parameters of the two instances are statically matching subtypes of the same type; and

14/5 - the output generated by Element_Type'Output or Element_Type'Write is readable by Element_Type'Input or Element_Type'Read, respectively (where Element_Type denotes the type of the two actual Element_Type parameters).

Implementation Advice

15/5 {*AI12-0254-1*} Bounded holder objects should be implemented without dynamic allocation and any finalization should be trivial unless Element_Type needs finalization.

15.a.1/5 **Implementation Advice:** Bounded holder objects should be implemented without dynamic allocation.

15.a/5 **To be honest:** Implementation of this container in Ada will probably require the use of a special storage pool. When we say "without dynamic allocation", we mean that this pool does not use heap memory and has a trivial finalization routine (that is, procedures Adjust and Finalize are null procedures). All storage pools are controlled, so we can't reasonably say that a bounded holder will not need finalization.

15.b/5 **Implementation Note:** If the implementation supports discontiguous objects that require multiple calls to Allocate in a storage pool, the storage pool will need to support such allocations. The storage pool implementation can assume that all Allocate calls occur together, and similarly for Deallocate calls, thus simplifying the pool implementation so that allocation only occurs at a high-water mark location.

16/5 {*AI12-0350-1*} {*AI12-0445-1*} The Implementation Advice about the Move and Swap operations is deleted for bounded holders; these operations can copy elements as necessary.

16.a/5 **Reason:** The memory of a bounded indefinite holder belongs directly to the container, so it cannot be moved with the element. If the element type contains any internal pointers, moving it without calling Adjust would leave such pointers pointing to the wrong holder object. Thus, a full copy is needed, including any associated finalization and adjustments.

Extensions to Ada 2012

16.b/5 {*AI12-0254-1*} The generic package Containers.Bounded_Indefinite_Holders is new.

A.18.33 Example of Container Use

Examples

{*AI05-0212-1*} The following example is an implementation of Dijkstra's shortest path algorithm in a directed graph with positive distances. The graph is represented by a map from nodes to sets of edges. 1/3

```
with Ada.Containers.Vectors;                                                    2/3
with Ada.Containers.Doubly_Linked_Lists;
use Ada.Containers;
generic
   type Node is range <>;
package Shortest_Paths is
   type Distance is new Float range 0.0 .. Float'Last;
   type Edge is record
      To, From : Node;
      Length   : Distance;
   end record;

   package Node_Maps is new Vectors (Node, Node);                               3/3
   -- The algorithm builds a map to indicate the node used to reach a given
   -- node in the shortest distance.

   package Adjacency_Lists is new Doubly_Linked_Lists (Edge);                   4/3
   use Adjacency_Lists;

   package Graphs is new Vectors (Node, Adjacency_Lists.List);                  5/3

   package Paths is new Doubly_Linked_Lists (Node);                             6/3

   function Shortest_Path                                                       7/3
     (G : Graphs.Vector; Source : Node; Target : Node) return Paths.List
      with Pre => G (Source) /= Adjacency_Lists.Empty_List;

end Shortest_Paths;                                                             8/3

{AI12-0178-1} package body Shortest_Paths is                                    9/5
   function Shortest_Path
     (G : Graphs.Vector; Source : Node; Target : Node) return Paths.List
   is
      use Node_Maps, Paths, Graphs;
      Reached  : array (Node) of Boolean := (others => False);
      -- The set of nodes whose shortest distance to the source is known.

{AI05-0299-1}       Reached_From : array (Node) of Node;                        10/3
      So_Far   : array (Node) of Distance := (others => Distance'Last);
      The_Path : Paths.List := Paths.Empty_List;
      Nearest_Distance : Distance;
      Next     : Node;
   begin
      So_Far(Source)  := 0.0;

      while not Reached(Target) loop                                            11/3
         Nearest_Distance := Distance'Last;

         -- Find closest node not reached yet, by iterating over all nodes.     12/3
         -- A more efficient algorithm uses a priority queue for this step.

         Next := Source;                                                        13/3
         for N in Node'First .. Node'Last loop
            if not Reached(N)
              and then So_Far(N) < Nearest_Distance then
                 Next := N;
                 Nearest_Distance := So_Far(N);
            end if;
         end loop;

{AI05-0299-1}          if Nearest_Distance = Distance'Last then                 14/3
            -- No next node found, graph is not connected
            return Paths.Empty_List;
```

```
15/3              else
                     Reached(Next) := True;
                  end if;
16/3              -- Update minimum distance to newly reachable nodes.
17/3        {AI05-0299-1}          for E of G (Next) loop
                  if not Reached(E.To) then
                     Nearest_Distance := E.Length + So_Far(Next);
18/3                 if Nearest_Distance < So_Far(E.To) then
                        Reached_From(E.To) := Next;
                        So_Far(E.To) := Nearest_Distance;
                     end if;
                  end if;
               end loop;
            end loop;
19/3        -- Rebuild path from target to source.
20/5        {AI12-0386-1}       declare
               N : Node := Target;
            begin
               Prepend (The_Path, N);
               while N /= Source loop
                  N := Reached_From(N);
                  Prepend (The_Path, N);
               end loop;
            end;
21/3        return The_Path;
         end;
      end Shortest_Paths;
```

22/3 {*AI05-0212-1*} Note that the effect of the Constant_Indexing aspect (on type Vector) and the Implicit_Dereference aspect (on type Reference_Type) is that

```
23/3    G (Next)
```

24/5 {*AI05-0212-1*} {*AI12-0426-1*} is a convenient shorthand for

```
25/3    G.Constant_Reference (Next).Element.all
```

26/3 {*AI05-0212-1*} Similarly, the effect of the loop:

```
27/3    for E of G (Next) loop
           if not Reached(E.To) then
              ...
           end if;
        end loop;
```

28/3 {*AI05-0212-1*} is the same as:

```
29/4    {AI12-0080-1} for C in G (Next).Iterate loop
           declare
              E : Edge renames G (Next)(C);
           begin
              if not Reached(E.To) then
                 ...
              end if;
           end;
        end loop;
```

{*AI05-0212-1*} which is the same as: 30/3

```
{AI12-0080-1} declare                                                    31/4
   L : Adjacency_Lists.List renames G (Next);
   C : Adjacency_Lists.Cursor := L.First;
begin
   while Has_Element (C) loop
      declare
         E : Edge renames L(C);
      begin
         if not Reached(E.To) then
            ...
         end if;
      end;
      C := L.Next (C);
   end loop;
end;
```

Wording Changes from Ada 2005

{*AI05-0212-1*} This example of container use is new. 31.a/3

A.19 The Package Locales

{*AI05-0127-2*} {*AI05-0248-1*} A *locale* identifies a geopolitical place or region and its associated language, which can be used to determine other internationalization-related characteristics. 1/3

Static Semantics

{*AI05-0127-2*} The library package Locales has the following declaration: 2/3

```
{AI12-0414-1} package Ada.Locales                                        3/5
   with Preelaborate, Remote_Types is

{AI12-0037-1}    type Language_Code is new String (1 .. 3)               4/4
      with Dynamic_Predicate =>
         (for all E of Language_Code => E in 'a' .. 'z');
   type Country_Code is new String (1 .. 2)
      with Dynamic_Predicate =>
         (for all E of Country_Code  => E in 'A' .. 'Z');
```

Discussion: {*AI12-0037-1*} These types are derived from type String so that they can easily be converted to or from type String. That's important if one of these values needs to be input or displayed (via Text_IO, perhaps). We use the predicate to ensure that only possible component values are used. Ada does not allow converting between unrelated types with components that don't statically match, so we cannot declare new types with constrained components if we want conversions to or from type String. 4.a/4

```
   Language_Unknown : constant Language_Code := "und";                   5/3
   Country_Unknown : constant Country_Code := "ZZ";

   function Language return Language_Code;                               6/3
   function Country return Country_Code;

end Ada.Locales;                                                         7/3
```

{*AI05-0127-2*} {*AI05-0233-1*} The *active locale* is the locale associated with the partition of the current task. 8/3

Implementation Note: {*AI05-0233-1*} Some environments define both a system locale and the locale of the current user. For such environments, the active locale is that of current user if any; otherwise (as in a partition running on a server without a user), the system locale should be used. 8.a/3

This paragraph was deleted.{*AI05-0127-2*} {*AI12-0446-1*} 9/5

This paragraph was deleted.{*AI05-0127-2*} {*AI12-0446-1*} 10/5

{*AI05-0127-2*} {*AI05-0248-1*} {*AI12-0446-1*} Function Language returns the code of the language associated with the active locale. If the Language_Code associated with the active locale cannot be 11/5

determined from the environment, then Language returns Language_Unknown. Otherwise, the result is a lower-case string representation of an ISO 639-3:2007 alpha-3 code that identifies a language.

11.a/3 **Discussion:** Some common language codes are: "eng" – English; "fra" – French; "deu" – German; "zho" – Chinese. These are the same codes as used by POSIX systems. We considered including constants for the most common languages, but that was rejected as the likely source of continual arguments about the constant names and which languages are important enough to include.

12/5 {*AI05-0127-2*} {*AI05-0248-1*} {*AI12-0446-1*} {*AI12-0450-1*} Function Country returns the code of the country associated with the active locale. If the Country_Code associated with the active locale cannot be determined from the environment, then Country returns Country_Unknown. Otherwise, the result is an upper-case string representation of an ISO 3166-1:2020 alpha-2 code that identifies a country.

12.a/3 **Discussion:** Some common country codes are: "CA" – Canada; "FR" – France; "DE" – Germany; "IT" – Italy; "ES" – Spain; "GB" – United Kingdom; "US" – United States. These are the same codes as used by POSIX systems. We didn't include any country constants for the same reasons that we didn't include any language constants.

Extensions to Ada 2005

12.b/3 {*AI05-0127-2*} {*AI05-0233-1*} Package Locales is new.

Inconsistencies With Ada 2012

12.c/4 {*AI12-0037-1*} **Corrigendum:** Types Language_Code and Country_Code are defined with predicates rather than constrained components so that they can be converted to/from type String. This changes the exception raised from Constraint_Error to Assertion_Error if an assignment is attempted with an incorrect value. This could only matter if there is a handler specifically for Constraint_Error surrounding this assignment; as this exception raise is certainly caused by a bug (why would anyone want to use invalid language or country codes?), such a handler seems very unlikely. (In addition, this is a new Ada 2012 package, so there is not likely to be a lot of code using it.)

Annex B
(normative)
Interface to Other Languages

This Annex describes features for writing mixed-language programs. General interface support is presented first; then specific support for C, COBOL, and Fortran is defined, in terms of language interface packages for each of these languages. 1

Ramification: This Annex is not a "Specialized Needs" annex. Every implementation must support all nonoptional features defined here (mainly the package Interfaces). 1.a

Language Design Principles

Ada should have strong support for mixed-language programming. 1.b

Implementation Requirements

{*AI05-0229-1*} {*AI05-0262-1*} {*AI05-0299-1*} Support for interfacing to any foreign language is optional. However, an implementation shall not provide any optional aspect, attribute, library unit, or pragma having the same name as an aspect, attribute, library unit, or pragma (respectively) specified in the subclauses of this Annex unless the provided construct is either as specified in those subclauses or is more limited in capability than that required by those subclauses. A program that attempts to use an unsupported capability of this Annex shall either be identified by the implementation before run time or shall raise an exception at run time. 2/3

Discussion: The intent is that the same rules apply for the optional parts of language interfacing as apply for Specialized Needs Annexes. See 1.1.3 for a discussion of the purpose of these rules. 2.a/3

Extensions to Ada 83

Much of the functionality in this Annex is new to Ada 95. 2.b

Wording Changes from Ada 83

This Annex contains what used to be RM83-13.8. 2.c

Wording Changes from Ada 2005

{*AI05-0262-1*} Moved the clarification that interfacing to foreign languages is optional and has the same restrictions as a Specialized Needs Annex here. 2.d/3

B.1 Interfacing Aspects

{*AI05-0229-1*} An *interfacing* aspect is a representation aspect that is one of the aspects Import, Export, Link_Name, External_Name, or Convention. 0.1/3

{*AI05-0229-1*} {*AI05-0269-1*} Specifying the Import aspect to have the value True is used to import an entity defined in a foreign language into an Ada program, thus allowing a foreign-language subprogram to be called from Ada, or a foreign-language variable to be accessed from Ada. In contrast, specifying the Export aspect to have the value True is used to export an Ada entity to a foreign language, thus allowing an Ada subprogram to be called from a foreign language, or an Ada object to be accessed from a foreign language. The Import and Export aspects are intended primarily for objects and subprograms, although implementations are allowed to support other entities. The Link_Name and External_Name aspects are used to specify the link name and external name, respectively, to be used to identify imported or exported entities in the external environment. 1/3

Aspect Description for Import: Entity is imported from another language. 1.a/3

1.b/3 **Aspect Description for Export:** Entity is exported to another language.

1.c/3 **Aspect Description for External_Name:** Name used to identify an imported or exported entity.

1.d/3 **Aspect Description for Link_Name:** Linker symbol used to identify an imported or exported entity.

2/3 {*AI05-0229-1*} The Convention aspect is used to indicate that an Ada entity should use the conventions of another language. It is intended primarily for types and "callback" subprograms. For example, "**with** Convention => Fortran" on the declaration of an array type Matrix implies that Matrix should be represented according to the conventions of the supported Fortran implementation, namely column-major order.

2.a/3 **Aspect Description for Convention:** Calling convention or other convention used for interfacing to other languages.

3/5 {*AI12-0445-1*} A pragma Linker_Options is used to specify the system linker parameters necessary when a given compilation unit is included in a partition.

Syntax

4/3 {*AI05-0229-1*} The form of a pragma Linker_Options is as follows:

Paragraphs 5 through 7 were moved to Annex J, "Obsolescent Features".

8 **pragma** Linker_Options(*string*_expression);

9 A pragma Linker_Options is allowed only at the place of a declarative_item.

9.1/3 *This paragraph was deleted.*{*8652/0058*} {*AI95-00036-01*} {*AI05-0229-1*}

Name Resolution Rules

9.2/3 {*AI05-0229-1*} The Import and Export aspects are of type Boolean.

10/3 {*AI05-0229-1*} The Link_Name and External_Name aspects are of type String.

10.a/3 **Ramification:** There is no language-defined support for external or link names of type Wide_String, or of other string types. Implementations may, of course, have additional aspects for that purpose. Note that allowing both String and Wide_String in the same aspect_definition would cause ambiguities.

10.1/3 {*AI05-0229-1*} The expected type for the *string*_expression in pragma Linker_Options is String.

Legality Rules

11/3 {*AI05-0229-1*} The aspect Convention shall be specified by a *convention*_identifier which shall be the name of a *convention*. The convention names are implementation defined, except for certain language-defined ones, such as Ada and Intrinsic, as explained in 6.3.1, "Conformance Rules". [Additional convention names generally represent the calling conventions of foreign languages, language implementations, or specific run-time models.] The convention of a callable entity is its *calling convention*.

11.a **Implementation defined:** Implementation-defined convention names.

11.b **Discussion:** We considered representing the convention names using an enumeration type declared in System. Then, *convention*_identifier would be changed to *convention*_name, and we would make its expected type be the enumeration type. We didn't do this because it seems to introduce extra complexity, and because the list of available languages is better represented as the list of children of package Interfaces — a more open-ended sort of list.

12 If *L* is a *convention*_identifier for a language, then a type T is said to be *compatible with convention L*, (alternatively, is said to be an *L-compatible type*) if any of the following conditions are met:

13 • T is declared in a language interface package corresponding to *L* and is defined to be *L*-compatible (see B.3, B.3.1, B.3.2, B.4, B.5),

14/3 • {*AI05-0229-1*} Convention *L* has been specified for T, and T is *eligible for convention L*; that is:

- {*AI12-0135-1*} T is an enumeration type such that all internal codes (whether assigned by default or explicitly) are within an implementation-defined range that includes at least the range of values 0 .. 2**15–1; 14.1/4
- T is an array type with either an unconstrained or statically-constrained first subtype, and its component type is *L*-compatible, 15
- T is a record type that has no discriminants and that only has components with statically-constrained subtypes, and each component type is *L*-compatible, 16
- {*AI05-0002-1*} T is an access-to-object type, its designated type is *L*-compatible, and its designated subtype is not an unconstrained array subtype, 17/3
- T is an access-to-subprogram type, and its designated profile's parameter and result types are all *L*-compatible. 18

- T is derived from an *L*-compatible type, 19
- {*AI12-0207-1*} T is an anonymous access type, and T is eligible for convention *L*, 19.1/5

  Reason: We say this so that the presence of an anonymous access component does not necessarily prevent a type from being eligible for convention *L*. We want the anonymous access type to take the convention from the enclosing type, but if we only said that, the definition would be circular (one can only portably apply the convention *L* to a record type R if the components of R already have convention *L*; but the anonymous components of R have to take the convention from R). We include the part of about T being eligible for convention *L* so that we don't force convention *L* on some type that is incompatible with it. 19.a/5

- The implementation permits T as an *L*-compatible type. 20

  Discussion: For example, an implementation might permit Integer as a C-compatible type, though the C type to which it corresponds might be different in different environments. 20.a

{*AI05-0229-1*} If the Convention aspect is specified for a type, then the type shall either be compatible with or eligible for the specified convention. 21/3

Ramification: {*AI05-0229-1*} If a type is derived from an *L*-compatible type, the derived type is by default *L*-compatible, but it is also permitted to specify the Convention aspect for the derived type. 21.a/3

{*AI05-0229-1*} It is permitted to specify the Convention aspect for an incomplete type, but in the complete declaration each component must be *L*-compatible. 21.b/3

{*AI05-0229-1*} If each component of a record type is *L*-compatible, then the record type itself is only *L*-compatible if it has a specified Convention. 21.c/3

{*AI12-0207-1*} If convention *L* is specified for a type T, for each component of T that has an anonymous access type, the convention of the anonymous access type is *L*. If convention *L* is specified for an object that has an anonymous access type, the convention of the anonymous access type is *L*. 21.1/5

Ramification: This applies to both anonymous access-to-object and anonymous access-to-subprogram types. 21.d/5

{*AI05-0229-1*} Notwithstanding any rule to the contrary, a declaration with a True Import aspect shall not have a completion. 22/3

Discussion: {*AI05-0229-1*} For declarations of deferred constants and subprograms, we explicitly mention that no completion is allowed when aspect Import is True. For other declarations that require completions, we ignore the possibility of the aspect Import being True. Nevertheless, if an implementation chooses to allow specifying aspect Import to be True for the declaration of a task, protected type, incomplete type, private type, etc., it may do so, and the normal completion is then not allowed for that declaration. 22.a/3

{*AI05-0229-1*} An entity with a True Import aspect (or Export aspect) is said to be *imported* (respectively, *exported*). An entity shall not be both imported and exported. 23/3

The declaration of an imported object shall not include an explicit initialization expression. [Default initializations are not performed.] 24

Proof: This follows from the “Notwithstanding ...” wording in the Dynamics Semantics paragraphs below. 24.a

25/3 {*AI05-0229-1*} The type of an imported or exported object shall be compatible with the specified Convention aspect, if any.

25.a **Ramification:** This implies, for example, that importing an Integer object might be illegal, whereas importing an object of type Interfaces.C.int would be permitted.

26/3 {*AI05-0229-1*} For an imported or exported subprogram, the result and parameter types shall each be compatible with the specified Convention aspect, if any.

27/3 {*AI05-0229-1*} The aspect_definition (if any) used to directly specify an Import, Export, External_Name, or Link_Name aspect shall be a static expression. The *string*_expression of a pragma Linker_Options shall be static. An External_Name or Link_Name aspect shall be specified only for an entity that is either imported or exported.

Static Semantics

Paragraphs 28 and 29 were deleted.

30/3 {*AI05-0229-1*} The Convention aspect represents the calling convention or representation convention of the entity. For an access-to-subprogram type, it represents the calling convention of designated subprograms. In addition:

31/3 - A True Import aspect indicates that the entity is defined externally (that is, outside the Ada program). This aspect is never inherited; if not directly specified, the Import aspect is False.

32/3 - A True Export aspect indicates that the entity is used externally. This aspect is never inherited; if not directly specified, the Export aspect is False.

33/3 - For an entity with a True Import or Export aspect, an external name, link name, or both may also be specified.

34 An *external name* is a string value for the name used by a foreign language program either for an entity that an Ada program imports, or for referring to an entity that an Ada program exports.

35 A *link name* is a string value for the name of an exported or imported entity, based on the conventions of the foreign language's compiler in interfacing with the system's linker tool.

36 The meaning of link names is implementation defined. If neither a link name nor the Address attribute of an imported or exported entity is specified, then a link name is chosen in an implementation-defined manner, based on the external name if one is specified.

36.a **Implementation defined:** The meaning of link names.

36.b **Ramification:** For example, an implementation might always prepend "_", and then pass it to the system linker.

36.c **Implementation defined:** The manner of choosing link names when neither the link name nor the address of an imported or exported entity is specified.

36.d **Ramification:** Normally, this will be the entity's defining name, or some simple transformation thereof.

37 Pragma Linker_Options has the effect of passing its string argument as a parameter to the system linker (if one exists), if the immediately enclosing compilation unit is included in the partition being linked. The interpretation of the string argument, and the way in which the string arguments from multiple Linker_Options pragmas are combined, is implementation defined.

37.a **Implementation defined:** The effect of pragma Linker_Options.

Dynamic Semantics

38/3 {*AI05-0229-1*} Notwithstanding what this document says elsewhere, the elaboration of a declaration with a True Import aspect does not create the entity. Such an elaboration has no other effect than to allow the defining name to denote the external entity.

Ramification: This implies that default initializations are skipped. (Explicit initializations are illegal.) For example, an imported access object is *not* initialized to **null**. 38.a

This paragraph was deleted.{*AI05-0229-1*} 38.b/3

Discussion: {*AI05-0229-1*} This “notwithstanding” wording is better than saying “unless aspect Import is True” on every definition of elaboration. It says we recognize the contradiction, and this rule takes precedence. 38.c/3

Erroneous Execution

{*AI95-00320-01*} {*AI05-0229-1*} {*AI12-0219-1*} It is the programmer's responsibility to ensure that the use of interfacing aspects does not violate Ada semantics; otherwise, program execution is erroneous. For example, passing an object with mode **in** to imported code that modifies it causes erroneous execution. Similarly, calling an imported subprogram that is not pure from a pure package causes erroneous execution. 38.1/5

Implementation Advice

{*AI05-0229-1*} If an implementation supports Export for a given language, then it should also allow the main subprogram to be written in that language. It should support some mechanism for invoking the elaboration of the Ada library units included in the system, and for invoking the finalization of the environment task. On typical systems, the recommended mechanism is to provide two subprograms whose link names are "adainit" and "adafinal". Adainit should contain the elaboration code for library units. Adafinal should contain the finalization code. These subprograms should have no effect the second and subsequent time they are called. 39/3

Implementation Advice: If Export is supported for a language, the main program should be able to be written in that language. Subprograms named "adainit" and "adafinal" should be provided for elaboration and finalization of the environment task. 39.a.1/3

Ramification: For example, if the main subprogram is written in C, it can call adainit before the first call to an Ada subprogram, and adafinal after the last. 39.a

{*AI05-0229-1*} {*AI05-0269-1*} Automatic elaboration of preelaborated packages should be provided when specifying the Export aspect as True is supported. 40/3

Implementation Advice: Automatic elaboration of preelaborated packages should be provided when specifying the Export aspect as True is supported. 40.a.1/3

{*AI05-0229-1*} {*AI12-0135-1*} {*AI12-0444-1*} For each supported convention *L* other than Intrinsic, an implementation should support specifying the Import and Export aspects for objects of *L*-compatible types and for subprograms, and the Convention aspect for *L*-eligible types and for subprograms, presuming the other language has corresponding features. Specifying the Convention aspect should be supported for enumeration types whose internal codes fall within the range 0 .. 2**15–1, but no recommendation is made for other scalar types. 41/5

Implementation Advice: For each supported convention *L* other than Intrinsic, specifying the aspects Import and Export should be supported for objects of *L*-compatible types and for subprograms, and aspect Convention should be supported for *L*-eligible types and for subprograms. 41.a.1/3

Reason: {*AI05-0229-1*} Specifying aspect Convention is not necessary for scalar types, since the language interface packages declare scalar types corresponding to those provided by the respective foreign languages. 41.a/3

Implementation Note: {*AI95-00114-01*} If an implementation supports interfacing to the C++ entities not supported by B.3, it should do so via the convention identifier C_Plus_Plus (in additional to any C++-implementation-specific ones). 41.b/2

Reason: {*AI95-00114-01*} The reason for giving the advice about C++ is to encourage uniformity among implementations, given that the name of the language is not syntactically legal as an identifier. 41.c/2

NOTE 1 {*AI05-0229-1*} {*AI12-0440-1*} Implementations can place restrictions on interfacing aspects; for example, requiring each exported entity to be declared at the library level. 42/5

Proof: Arbitrary restrictions are allowed by 13.1. 42.a

42.b **Ramification:** Such a restriction might be to disallow them altogether. Alternatively, the implementation might allow them only for certain kinds of entities, or only for certain conventions.

43/3 NOTE 2 {*AI05-0229-1*} The Convention aspect in combination with the Import aspect indicates the conventions for accessing external entities. It is possible that the actual entity is written in assembly language, but reflects the conventions of a particular language. For example, `with Convention => Ada` can be used to interface to an assembly language routine that obeys the Ada compiler's calling conventions.

44/5 NOTE 3 {*AI05-0229-1*} {*AI12-0440-1*} To obtain "call-back" to an Ada subprogram from a foreign language environment, the Convention aspect can be specified both for the access-to-subprogram type and the specific subprogram(s) to which 'Access is applied.

Paragraphs 45 and 46 were deleted.

47/5 NOTE 4 {*AI12-0449-1*} Machine code insertions can also be relevant for interfacing; see 13.8.

47.a/3 **Ramification:** {*AI05-0229-1*} The Intrinsic convention (see 6.3.1) implies that the entity is somehow "built in" to the implementation. Thus, it generally does not make sense for users to specify Intrinsic along with specifying that the entity is imported. The intention is that only implementations will specify Intrinsic for an imported entity. The language also defines certain subprograms to be Intrinsic.

47.b/3 **Discussion:** {*AI05-0229-1*} There are many imaginable interfacing aspects that don't make any sense. For example, setting the Convention of a protected procedure to Ada is probably wrong. Rather than enumerating all such cases, however, we leave it up to implementations to decide what is sensible.

48/3 NOTE 5 {*AI05-0229-1*} If both External_Name and Link_Name are specified for a given entity, then the External_Name is ignored.

49/2 *This paragraph was deleted.*{*AI95-00320-01*}

Examples

50/4 *{AI12-0080-1} Example of interfacing aspects:*

51/3
```
{AI05-0229-1} {AI05-0269-1} package Fortran_Library is
  function Sqrt (X : Float) return Float
    with Import => True, Convention => Fortran;
  type Matrix is array (Natural range <>, Natural range <>) of Float
    with Convention => Fortran;
  function Invert (M : Matrix) return Matrix
    with Import => True, Convention => Fortran;
end Fortran_Library;
```

Extensions to Ada 83

51.a Interfacing pragmas are new to Ada 95. Pragma Import replaces Ada 83's pragma Interface. Existing implementations can continue to support pragma Interface for upward compatibility.

Wording Changes from Ada 95

51.b/2 {*8652/0058*} {*AI95-00036-01*} **Corrigendum:** Clarified that pragmas Import and Export work like a subprogram call; parameters cannot be omitted unless named notation is used. (Reordering is still not permitted, however.)

51.c/2 {*AI95-00320-01*} Added wording to say all bets are off if foreign code doesn't follow the semantics promised by the Ada specifications.

Incompatibilities With Ada 2005

51.d/3 {*AI05-0002-1*} **Correction:** Access types that designate unconstrained arrays are no longer defined to be *L*-compatible. Such access-to-arrays require bounds information, which is likely to be incompatible with a foreign language. The change will allow (but not require) compilers to reject bad uses, which probably will not work anyway. Note that implementations can still support any type that it wants as *L*-compatible; such uses will not be portable, however. As such, there should be little existing code that will be impacted (compilers probably already rejected cases that could not be translated, whether or not the language allowed doing so formally).

Extensions to Ada 2005

51.e/3 {*AI05-0229-1*} Aspects Convention, Import, Export, Link_Name, and External_Name are new; pragmas Convention, Import, and Export are now obsolescent.

Incompatibilities With Ada 2012

{*AI12-0207-1*} **Correction:** The convention of anonymous access components is that of the enclosing type (in Ada 2012, it was Ada). Similarly, the convention of the anonymous access type of an object is that of the object (again, in Ada 2012 it was Ada). While this is formally incompatible, it should be more useful in portable code; it makes little sense to have a component of an Ada access type in a record with a C convention. For most implementations, this will have no real effect as convention Ada anonymous access types were allowed as C-compatible anyway. But such code was not portable, as this was not required in Ada 2012. 51.f/5

Extensions to Ada 2012

{*AI12-0135-1*} **Corrigendum:** Added a suggestion that convention be supported for enumeration types. This will make the use of enumeration types portable for implementations that support interfacing to a particular language. 51.g/4

Wording Changes from Ada 2012

{*AI12-0219-1*} **Correction:** Added some examples to the erroneous execution text; this is a very important rule as it means that Ada compilers can assume that provided interfacing declarations reflect the actual foreign code. 51.h/5

B.2 The Package Interfaces

Package Interfaces is the parent of several library packages that declare types and other entities useful for interfacing to foreign languages. It also contains some implementation-defined types that are useful across more than one language (in particular for interfacing to assembly language). 1

Implementation defined: The contents of the visible part of package Interfaces and its language-defined descendants. 1.a

Static Semantics

The library package Interfaces has the following skeletal declaration: 2

{*AI12-0414-1*} 3/5

```
package Interfaces
   with Pure is

   type Integer_n is range -2**(n-1) .. 2**(n-1) - 1;  --2's complement

   type Unsigned_n is mod 2**n;

   function Shift_Left  (Value : Unsigned_n; Amount : Natural)
      return Unsigned_n;
   function Shift_Right (Value : Unsigned_n; Amount : Natural)
      return Unsigned_n;
   function Shift_Right_Arithmetic (Value : Unsigned_n; Amount : Natural)
      return Unsigned_n;
   function Rotate_Left  (Value : Unsigned_n; Amount : Natural)
      return Unsigned_n;
   function Rotate_Right (Value : Unsigned_n; Amount : Natural)
      return Unsigned_n;
   ...
end Interfaces;
```

4 5 6

Implementation Requirements

An implementation shall provide the following declarations in the visible part of package Interfaces: 7

- Signed and modular integer types of *n* bits, if supported by the target architecture, for each *n* that is at least the size of a storage element and that is a factor of the word size. The names of these types are of the form Integer_*n* for the signed types, and Unsigned_*n* for the modular types; 8

  Ramification: For example, for a typical 32-bit machine the corresponding types might be Integer_8, Unsigned_8, Integer_16, Unsigned_16, Integer_32, and Unsigned_32. 8.a

  The wording above implies, for example, that Integer_16'Size = Unsigned_16'Size = 16. Unchecked conversions between same-Sized types will work as expected. 8.b

- For each such modular type in Interfaces, shifting and rotating subprograms as specified in the declaration of Interfaces above. These subprograms are Intrinsic. They operate on a bit-by-bit basis, using the binary representation of the value of the operands to yield a binary representation 9

for the result. The Amount parameter gives the number of bits by which to shift or rotate. For shifting, zero bits are shifted in, except in the case of Shift_Right_Arithmetic, where one bits are shifted in if Value is at least half the modulus.

9.a **Reason:** We considered making shifting and rotating be primitive operations of all modular types. However, it is a design principle of Ada that all predefined operations should be operators (not functions named by identifiers). (Note that an early version of Ada had "**abs**" as an identifier, but it was changed to a reserved word operator before standardization of Ada 83.) This is important because the implicit declarations would hide nonoverloadable declarations with the same name, whereas operators are always overloadable. Therefore, we would have had to make shift and rotate into reserved words, which would have been upward incompatible, or else invent new operator symbols, which seemed like too much mechanism.

9.b/5 **To be honest:** {*AI12-0264-1*} "Shifting" and "rotating" have the conventional meaning. Neither of these terms is usefully defined by the usual normative references of the Reference Manual, so we provide pseudo-code here to describe the intended semantics of the above wording (all operations in these examples are using modular semantics).

9.c/5
```
function Rotate_Left (Value : Unsigned_n; Amount : Natural)
   return Unsigned_n is
   Result : Unsigned_n := Value;
   Bit : Unsigned_n range 0 .. 1;
begin
   for Count in 1 .. Amount loop
      Bit := Result/2**(n-1); -- High-bit of Result
      Result := Result*2 + Bit;
   end loop;
   return Result;
end Rotate_Left;
```

9.d/5
```
function Rotate_Right (Value : Unsigned_n; Amount : Natural)
   return Unsigned_n is
   Result : Unsigned_n := Value;
   Bit : Unsigned_n range 0 .. 1;
begin
   for Count in 1 .. Amount loop
      Bit := Result mod 2; -- Low-bit of Result
      Result := Result/2 + (Bit * 2**(n-1));
   end loop;
   return Result;
end Rotate_Right;
```

9.e/5
```
function Shift_Left (Value : Unsigned_n; Amount : Natural)
   return Unsigned_n is
   Result : Unsigned_n := Value;
begin
   for Count in 1 .. Amount loop
      Result := Result * 2;
   end loop;
   return Result;
end Shift_Left;
```

9.f/5
```
function Shift_Right (Value : Unsigned_n; Amount : Natural)
   return Unsigned_n is
   Result : Unsigned_n := Value;
begin
   for Count in 1 .. Amount loop
      Result := Result / 2;
   end loop;
   return Result;
end Shift_Right;
```

9.g/5
```
function Shift_Right_Arithmetic (Value : Unsigned_n; Amount : Natural)
   return Unsigned_n is
   Result : Unsigned_n := Value;
   Neg : constant Boolean :=
      Result/2**(n-1) = 1; -- High-bit of Result
begin
   for Count in 1 .. Amount loop
      if Neg then
         Result := Result / 2 + 2**(n-1);
      else
         Result := Result / 2;
      end if;
   end loop;
   return Result;
end Shift_Right_Arithmetic;
```

{*AI12-0264-1*} These generally correspond to machine instructions, although there may not be an exact match in terms of boundary conditions, as Ada requires the correct result to be produced for all values of Amount. 9.h/5

- Floating point types corresponding to each floating point format fully supported by the hardware. 10

Implementation Note: The names for these floating point types are not specified. However, if IEEE arithmetic is supported, then the names should be IEEE_Float_32 and IEEE_Float_64 for single and double precision, respectively. 10.a

Implementation Permissions

An implementation may provide implementation-defined library units that are children of Interfaces, and may add declarations to the visible part of Interfaces in addition to the ones defined above. 11

Implementation defined: Implementation-defined children of package Interfaces. 11.a/2

{*AI95-00204-01*} {*AI05-0229-1*} A child package of package Interfaces with the name of a convention may be provided independently of whether the convention is supported by the Convention aspect and vice versa. Such a child package should contain any declarations that would be useful for interfacing to the language (implementation) represented by the convention. Any declarations useful for interfacing to any language on the given hardware architecture should be provided directly in Interfaces. 11.1/3

Ramification: For example, package Interfaces.XYZ_Pascal might contain declarations of types that match the data types provided by the XYZ implementation of Pascal, so that it will be more convenient to pass parameters to a subprogram whose convention is XYZ_Pascal. 11.b/2

Implementation Advice

This paragraph was deleted.{*AI95-00204-01*} 12/2

This paragraph was deleted. 12.a/2

{*AI05-0299-1*} An implementation supporting an interface to C, COBOL, or Fortran should provide the corresponding package or packages described in the following subclauses. 13/3

Implementation Advice: If an interface to C, COBOL, or Fortran is provided, the corresponding package or packages described in Annex B, “Interface to Other Languages” should also be provided. 13.a.1/2

Implementation Note: The intention is that an implementation might support several implementations of the foreign language: Interfaces.This_Fortran and Interfaces.That_Fortran might both exist. The “default” implementation, overridable by the user, should be declared as a renaming: 13.a

```
package Interfaces.Fortran renames Interfaces.This_Fortran;
```
13.b

Wording Changes from Ada 95

{*AI95-00204-01*} Clarified that interfacing to foreign languages is optional and has the same restrictions as a Specialized Needs Annex. 13.c/2

Wording Changes from Ada 2005

{*AI05-0262-1*} Move the restrictions on implementations of optional features to the start of this Annex. 13.d/3

B.3 Interfacing with C and C++

{*8652/0059*} {*AI95-00131-01*} {*AI95-00376-01*} {*AI05-0229-1*} {*AI12-0028-1*} The facilities relevant to interfacing with the C language and the corresponding subset of the C++ language are the package Interfaces.C and its children, and support for specifying the Convention aspect with *convention*_identifiers C, C_Pass_By_Copy, and any of the C_Variadic_*n* conventions described below. 1/4

{*AI95-00376-01*} {*AI95-0262-1*} {*AI95-0299-1*} The package Interfaces.C contains the basic types, constants, and subprograms that allow an Ada program to pass scalars and strings to C and C++ functions. When this subclause mentions a C entity, the reference also applies to the corresponding entity in C++. 2/3

Static Semantics

3 The library package Interfaces.C has the following declaration:

4/5
```
{AI12-0414-1} package Interfaces.C
   with Pure is

   -- Declarations based on C's <limits.h>

   CHAR_BIT  : constant := implementation-defined;   -- typically 8
   SCHAR_MIN : constant := implementation-defined;   -- typically –128
   SCHAR_MAX : constant := implementation-defined;   -- typically 127
   UCHAR_MAX : constant := implementation-defined;   -- typically 255

   -- Signed and Unsigned Integers
   type int   is range implementation-defined;
   type short is range implementation-defined;
   type long  is range implementation-defined;

   type signed_char is range SCHAR_MIN .. SCHAR_MAX;
   for signed_char'Size use CHAR_BIT;

   type unsigned       is mod implementation-defined;
   type unsigned_short is mod implementation-defined;
   type unsigned_long  is mod implementation-defined;

   type unsigned_char is mod (UCHAR_MAX+1);
   for unsigned_char'Size use CHAR_BIT;

   subtype plain_char is implementation-defined;

   type ptrdiff_t is range implementation-defined;

   type size_t is mod implementation-defined;

{AI12-0411-1}     -- Boolean Type
   type C_bool is new Boolean;

   -- Floating Point

   type C_float     is digits implementation-defined;

   type double      is digits implementation-defined;

   type long_double is digits implementation-defined;

   -- Characters and Strings

   type char is <implementation-defined character type>;

{8652/0060} {AI95-00037-01}    nul : constant char := implementation-defined;

   function To_C   (Item : in Character) return char;

   function To_Ada (Item : in char) return Character;

{AI05-0229-1} {AI05-0269-1}    type char_array is array (size_t range <>) of
aliased char
      with Pack;
   for char_array'Component_Size use CHAR_BIT;

   function Is_Nul_Terminated (Item : in char_array) return Boolean;

   function To_C   (Item       : in String;
                    Append_Nul : in Boolean := True)
      return char_array;

   function To_Ada (Item     : in char_array;
                    Trim_Nul : in Boolean := True)
      return String;

   procedure To_C (Item       : in  String;
                   Target     : out char_array;
                   Count      : out size_t;
                   Append_Nul : in  Boolean := True);

   procedure To_Ada (Item     : in  char_array;
                     Target   : out String;
                     Count    : out Natural;
                     Trim_Nul : in  Boolean := True);
```

```
   -- Wide Character and Wide String                                                    29
{8652/0060} {AI95-00037-01}    type wchar_t is <implementation-defined character type>;  30/1
{8652/0060} {AI95-00037-01}    wide_nul : constant wchar_t := implementation-defined;    31/1
   function To_C   (Item : in Wide_Character) return wchar_t;                           32
   function To_Ada (Item : in wchar_t       ) return Wide_Character;
{AI05-0229-1}    type wchar_array is array (size_t range <>) of aliased wchar_t        33/3
      with Pack;
This paragraph was deleted.{AI05-0229-1}                                               34/3
   function Is_Nul_Terminated (Item : in wchar_array) return Boolean;                    35
   function To_C   (Item       : in Wide_String;                                         36
                    Append_Nul : in Boolean := True)
      return wchar_array;
   function To_Ada (Item     : in wchar_array;                                           37
                    Trim_Nul : in Boolean := True)
      return Wide_String;
   procedure To_C (Item       : in  Wide_String;                                        38
                   Target     : out wchar_array;
                   Count      : out size_t;
                   Append_Nul : in  Boolean := True);
   procedure To_Ada (Item     : in  wchar_array;                                        39
                     Target   : out Wide_String;
                     Count    : out Natural;
                     Trim_Nul : in  Boolean := True);
{AI95-00285-01} {AI12-0450-1}    -- ISO/IEC 10646 compatible types.                    39.1/5
{AI95-00285-01}    type char16_t is <implementation-defined character type>;          39.2/2
   char16_nul : constant char16_t := implementation-defined;                          39.3/2
   function To_C (Item : in Wide_Character) return char16_t;                          39.4/2
   function To_Ada (Item : in char16_t) return Wide_Character;
{AI05-0229-1}    type char16_array is array (size_t range <>) of aliased char16_t      39.5/3
      with Pack;
This paragraph was deleted.{AI05-0229-1}                                              39.6/3
   function Is_Nul_Terminated (Item : in char16_array) return Boolean;                39.7/2
   function To_C (Item       : in Wide_String;
                  Append_Nul : in Boolean := True)
      return char16_array;
   function To_Ada (Item     : in char16_array;                                       39.8/2
                    Trim_Nul : in Boolean := True)
      return Wide_String;
   procedure To_C (Item       : in  Wide_String;                                     39.9/2
                   Target     : out char16_array;
                   Count      : out size_t;
                   Append_Nul : in  Boolean := True);
   procedure To_Ada (Item     : in  char16_array;                                    39.10/2
                     Target   : out Wide_String;
                     Count    : out Natural;
                     Trim_Nul : in  Boolean := True);
{AI95-00285-01}    type char32_t is <implementation-defined character type>;          39.11/2
   char32_nul : constant char32_t := implementation-defined;                          39.12/2
   function To_C (Item : in Wide_Wide_Character) return char32_t;                     39.13/2
   function To_Ada (Item : in char32_t) return Wide_Wide_Character;
{AI05-0229-1}    type char32_array is array (size_t range <>) of aliased char32_t      39.14/3
      with Pack;
This paragraph was deleted.{AI05-0229-1}                                              39.15/3
```

39.16/2
```
function Is_Nul_Terminated (Item : in char32_array) return Boolean;
function To_C (Item       : in Wide_Wide_String;
               Append_Nul : in Boolean := True)
   return char32_array;
```

39.17/2
```
function To_Ada (Item     : in char32_array;
                 Trim_Nul : in Boolean := True)
   return Wide_Wide_String;
```

39.18/2
```
procedure To_C (Item       : in  Wide_Wide_String;
                Target     : out char32_array;
                Count      : out size_t;
                Append_Nul : in  Boolean := True);
```

39.19/2
```
procedure To_Ada (Item     : in  char32_array;
                  Target   : out Wide_Wide_String;
                  Count    : out Natural;
                  Trim_Nul : in  Boolean := True);
```

40
```
Terminator_Error : exception;
```

41
```
end Interfaces.C;
```

41.a.1/2 **Implementation defined:** The definitions of certain types and constants in Interfaces.C.

42 Each of the types declared in Interfaces.C is C-compatible.

43/5 {*AI95-00285-01*} {*AI12-0411-1*} The types int, short, long, unsigned, ptrdiff_t, size_t, double, char, wchar_t, char16_t, and char32_t correspond respectively to the C types having the same names. The types signed_char, unsigned_short, unsigned_long, unsigned_char, C_bool, C_float, and long_double correspond respectively to the C types signed char, unsigned short, unsigned long, unsigned char, bool, float, and long double.

43.a/2 **Discussion:** The C types wchar_t and char16_t seem to be the same. However, wchar_t has an implementation-defined size, whereas char16_t is guaranteed to be an unsigned type of at least 16 bits. Also, char16_t and char32_t are encouraged to have UTF-16 and UTF-32 representations; that means that they are not directly the same as the Ada types, which most likely don't use any UTF encoding.

44 The type of the subtype plain_char is either signed_char or unsigned_char, depending on the C implementation.

45
```
function To_C   (Item : in Character) return char;
function To_Ada (Item : in char     ) return Character;
```

46 The functions To_C and To_Ada map between the Ada type Character and the C type char.

46.a.1/1 **Implementation Note:** {*8652/0114*} {*AI95-00038-01*} The To_C and To_Ada functions map between corresponding characters, not necessarily between characters with the same internal representation. Corresponding characters are characters defined by the same enumeration literal, if such exist; otherwise, the correspondence is unspecified.

46.a.2/1 The following definition is equivalent to the above summary:

46.a.3/1 `To_C (Latin_1_Char) = char'Value(Character'Image(Latin_1_Char))`
provided that char'Value does not raise an exception; otherwise the result is unspecified.

46.a.4/1 `To_Ada (Native_C_Char) = Character'Value(char'Image(Native_C_Char))`
provided that Character'Value does not raise an exception; otherwise the result is unspecified.

47
```
function Is_Nul_Terminated (Item : in char_array) return Boolean;
```

48 The result of Is_Nul_Terminated is True if Item contains nul, and is False otherwise.

49
```
function To_C   (Item : in String;     Append_Nul : in Boolean := True)
   return char_array;

function To_Ada (Item : in char_array; Trim_Nul   : in Boolean := True)
   return String;
```

50/2 {*AI95-00258-01*} The result of To_C is a char_array value of length Item'Length (if Append_Nul is False) or Item'Length+1 (if Append_Nul is True). The lower bound is 0. For each component Item(I), the corresponding component in the result is To_C applied to Item(I). The value nul is

appended if Append_Nul is True. If Append_Nul is False and Item'Length is 0, then To_C propagates Constraint_Error.

The result of To_Ada is a String whose length is Item'Length (if Trim_Nul is False) or the length of the slice of Item preceding the first nul (if Trim_Nul is True). The lower bound of the result is 1. If Trim_Nul is False, then for each component Item(I) the corresponding component in the result is To_Ada applied to Item(I). If Trim_Nul is True, then for each component Item(I) before the first nul the corresponding component in the result is To_Ada applied to Item(I). The function propagates Terminator_Error if Trim_Nul is True and Item does not contain nul. 51

```
procedure To_C (Item       : in  String;                                    52
                Target     : out char_array;
                Count      : out size_t;
                Append_Nul : in  Boolean := True);

procedure To_Ada (Item     : in  char_array;
                  Target   : out String;
                  Count    : out Natural;
                  Trim_Nul : in  Boolean := True);
```

For procedure To_C, each element of Item is converted (via the To_C function) to a char, which is assigned to the corresponding element of Target. If Append_Nul is True, nul is then assigned to the next element of Target. In either case, Count is set to the number of Target elements assigned. If Target is not long enough, Constraint_Error is propagated. 53

For procedure To_Ada, each element of Item (if Trim_Nul is False) or each element of Item preceding the first nul (if Trim_Nul is True) is converted (via the To_Ada function) to a Character, which is assigned to the corresponding element of Target. Count is set to the number of Target elements assigned. If Target is not long enough, Constraint_Error is propagated. If Trim_Nul is True and Item does not contain nul, then Terminator_Error is propagated. 54

```
function Is_Nul_Terminated (Item : in wchar_array) return Boolean;          55
```

The result of Is_Nul_Terminated is True if Item contains wide_nul, and is False otherwise. 56

```
function To_C   (Item : in Wide_Character) return wchar_t;                  57
function To_Ada (Item : in wchar_t       ) return Wide_Character;
```

To_C and To_Ada provide the mappings between the Ada and C wide character types. 58

```
function To_C   (Item       : in Wide_String;                               59
                 Append_Nul : in Boolean := True)
   return wchar_array;

function To_Ada (Item     : in wchar_array;
                 Trim_Nul : in Boolean := True)
   return Wide_String;

procedure To_C (Item       : in  Wide_String;
                Target     : out wchar_array;
                Count      : out size_t;
                Append_Nul : in  Boolean := True);

procedure To_Ada (Item     : in  wchar_array;
                  Target   : out Wide_String;
                  Count    : out Natural;
                  Trim_Nul : in  Boolean := True);
```

The To_C and To_Ada subprograms that convert between Wide_String and wchar_array have analogous effects to the To_C and To_Ada subprograms that convert between String and char_array, except that wide_nul is used instead of nul. 60

60.1/2
```
function Is_Nul_Terminated (Item : in char16_array) return Boolean;
```

60.2/2 {*AI95-00285-01*} The result of Is_Nul_Terminated is True if Item contains char16_nul, and is False otherwise.

60.3/2
```
function To_C (Item : in Wide_Character) return char16_t;
function To_Ada (Item : in char16_t ) return Wide_Character;
```

60.4/2 {*AI95-00285-01*} To_C and To_Ada provide mappings between the Ada and C 16-bit character types.

60.5/2
```
function To_C (Item       : in Wide_String;
               Append_Nul : in Boolean := True)
   return char16_array;

function To_Ada (Item     : in char16_array;
                 Trim_Nul : in Boolean := True)
   return Wide_String;

procedure To_C (Item       : in  Wide_String;
                Target     : out char16_array;
                Count      : out size_t;
                Append_Nul : in  Boolean := True);

procedure To_Ada (Item     : in  char16_array;
                  Target   : out Wide_String;
                  Count    : out Natural;
                  Trim_Nul : in  Boolean := True);
```

60.6/2 {*AI95-00285-01*} The To_C and To_Ada subprograms that convert between Wide_String and char16_array have analogous effects to the To_C and To_Ada subprograms that convert between String and char_array, except that char16_nul is used instead of nul.

60.7/2
```
function Is_Nul_Terminated (Item : in char32_array) return Boolean;
```

60.8/5 {*AI95-00285-01*} {*AI12-0437-1*} The result of Is_Nul_Terminated is True if Item contains char32_nul, and is False otherwise.

60.9/2
```
function To_C (Item : in Wide_Wide_Character) return char32_t;
function To_Ada (Item : in char32_t ) return Wide_Wide_Character;
```

60.10/2 {*AI95-00285-01*} To_C and To_Ada provide mappings between the Ada and C 32-bit character types.

60.11/2
```
function To_C (Item       : in Wide_Wide_String;
               Append_Nul : in Boolean := True)
   return char32_array;

function To_Ada (Item     : in char32_array;
                 Trim_Nul : in Boolean := True)
   return Wide_Wide_String;

procedure To_C (Item       : in  Wide_Wide_String;
                Target     : out char32_array;
                Count      : out size_t;
                Append_Nul : in  Boolean := True);

procedure To_Ada (Item     : in  char32_array;
                  Target   : out Wide_Wide_String;
                  Count    : out Natural;
                  Trim_Nul : in  Boolean := True);
```

60.12/2 {*AI95-00285-01*} The To_C and To_Ada subprograms that convert between Wide_Wide_String and char32_array have analogous effects to the To_C and To_Ada subprograms that convert between String and char_array, except that char32_nul is used instead of nul.

Discussion: The Interfaces.C package provides an implementation-defined character type, char, designed to model the C run-time character set, and mappings between the types char and Character. 60.a

One application of the C interface package is to compose a C string and pass it to a C function. One way to do this is for the programmer to declare an object that will hold the C array, and then pass this array to the C function. This is realized via the type char_array: 60.b

```
type char_array is array (size_t range <>) of Char;
```
60.c

The programmer can declare an Ada String, convert it to a char_array, and pass the char_array as actual parameter to the C function that is expecting a char *. 60.d

An alternative approach is for the programmer to obtain a C char pointer from an Ada String (or from a char_array) by invoking an allocation function. The package Interfaces.C.Strings (see below) supplies the needed facilities, including a private type chars_ptr that corresponds to C's char *, and two allocation functions. To avoid storage leakage, a Free procedure releases the storage that was allocated by one of these allocate functions. 60.e

It is typical for a C function that deals with strings to adopt the convention that the string is delimited by a nul char. The C interface packages support this convention. A constant nul of type Char is declared, and the function Value(Chars_Ptr) in Interfaces.C.Strings returns a char_array up to and including the first nul in the array that the chars_ptr points to. The Allocate_Chars function allocates an array that is nul terminated. 60.f

Some C functions that deal with strings take an explicit length as a parameter, thus allowing strings to be passed that contain nul as a data element. Other C functions take an explicit length that is an upper bound: the prefix of the string up to the char before nul, or the prefix of the given length, is used by the function, whichever is shorter. The C Interface packages support calling such functions. 60.g

{*8652/0059*} {*AI95-00131-01*} {*AI05-0229-1*} The Convention aspect with *convention*_identifier C_Pass_By_Copy shall only be specified for a type. 60.13/3

{*8652/0059*} {*AI95-00131-01*} {*AI95-00216-01*} The eligibility rules in B.1 do not apply to convention C_Pass_By_Copy. Instead, a type T is eligible for convention C_Pass_By_Copy if T is an unchecked union type or if T is a record type that has no discriminants and that only has components with statically constrained subtypes, and each component is C-compatible. 60.14/2

{*8652/0059*} {*AI95-00131-01*} {*AI05-0264-1*} If a type is C_Pass_By_Copy-compatible, then it is also C-compatible. 60.15/3

{*AI12-0028-1*} The identifiers C_Variadic_0, C_Variadic_1, C_Variadic_2, and so on are *convention*_identifiers. These conventions are said to be *C_Variadic*. The convention C_Variadic_*n* is the calling convention for a variadic C function taking *n* fixed parameters and then a variable number of additional parameters. The C_Variadic_*n* convention shall only be specified as the convention aspect for a subprogram, or for an access-to-subprogram type, having at least *n* parameters. A type is compatible with a C_Variadic convention if and only if the type is C-compatible. 60.16/4

To be honest: It is implementation defined what the largest *n* in C_Variadic_*n* is supported. We don't say this because it complicates the wording and it is true for almost any *convention*_identifier (only Ada is required to be supported by the language, all others need to be documented in order for programmers to know that they are available). 60.h/4

Implementation Requirements

{*8652/0059*} {*AI95-00131-01*} {*AI05-0229-1*} An implementation shall support specifying aspect Convention with a C *convention*_identifier for a C-eligible type (see B.1). An implementation shall support specifying aspect Convention with a C_Pass_By_Copy *convention*_identifier for a C_Pass_By_Copy-eligible type. 61/3

Implementation Permissions

An implementation may provide additional declarations in the C interface packages. 62

{*AI05-0002-1*} {*AI05-0229-1*} {*AI12-0444-1*} An implementation is not required to support specifying the Convention aspect with *convention*_identifier C in the following cases: 62.1/5

- {*AI05-0248-1*} for a subprogram that has a parameter of an unconstrained array subtype, unless the Import aspect has the value True for the subprogram; 62.2/3

62.3/3 • for a function with an unconstrained array result subtype;

62.4/3 • for an object whose nominal subtype is an unconstrained array subtype.

62.a/3 **Implementation Note:** {*AI05-0002-1*} These rules ensure that an implementation never needs to create bounds for an unconstrained array that originates in C (and thus does not have bounds). An implementation can do so if it wishes, of course. Note that these permissions do not extend to passing an unconstrained array as a parameter to a C function; in this case, the bounds can simply be dropped and thus support is required.

Implementation Advice

62.5/3 {*8652/0060*} {*AI95-00037-01*} {*AI95-00285-01*} The constants nul, wide_nul, char16_nul, and char32_nul should have a representation of zero.

62.b/2 **Implementation Advice:** The constants nul, wide_nul, char16_nul, and char32_nul in package Interfaces.C should have a representation of zero.

63 An implementation should support the following interface correspondences between Ada and C.

64 • An Ada procedure corresponds to a void-returning C function.

64.a **Discussion:** The programmer can also choose an Ada procedure when the C function returns an int that is to be discarded.

65 • An Ada function corresponds to a non-void C function.

65.1/4 • {*AI12-0135-1*} An Ada enumeration type corresponds to a C enumeration type with corresponding enumeration literals having the same internal codes, provided the internal codes fall within the range of the C int type.

66 • An Ada **in** scalar parameter is passed as a scalar argument to a C function.

67 • An Ada **in** parameter of an access-to-object type with designated type T is passed as a t* argument to a C function, where t is the C type corresponding to the Ada type T.

68 • An Ada **access** T parameter, or an Ada **out** or **in out** parameter of an elementary type T, is passed as a t* argument to a C function, where t is the C type corresponding to the Ada type T. In the case of an elementary **out** or **in out** parameter, a pointer to a temporary copy is used to preserve by-copy semantics.

68.1/2 • {*8652/0059*} {*AI95-00131-01*} {*AI95-00343-01*} An Ada parameter of a (record) type T of convention C_Pass_By_Copy, of mode **in**, is passed as a t argument to a C function, where t is the C struct corresponding to the Ada type T.

69/5 • {*8652/0059*} {*AI95-00131-01*} {*AI95-00343-01*} {*AI12-0219-1*} An Ada parameter of a record type T, other than an **in** parameter of a type of convention C_Pass_By_Copy, is passed as a t* argument to a C function, with the const modifier if the Ada mode is **in**, where t is the C struct corresponding to the Ada type T.

70/5 • {*AI12-0219-1*} An Ada parameter of an array type with component type T is passed as a t* argument to a C function, with the const modifier if the Ada mode is **in**, where t is the C type corresponding to the Ada type T.

71 • An Ada parameter of an access-to-subprogram type is passed as a pointer to a C function whose prototype corresponds to the designated subprogram's specification.

71.1/3 • {*AI05-0002-1*} An Ada parameter of a private type is passed as specified for the full view of the type.

71.2/3 • {*AI05-0002-1*} The rules of correspondence given above for parameters of mode **in** also apply to the return object of a function.

71.3/5 {*AI95-00337-01*} {*AI05-0002-1*} {*AI12-0184-1*} An implementation should provide unsigned_long_long and long_long as 64-bit modular and signed integer types (respectively) in package Interfaces.C if the C implementation supports unsigned long long and long long as 64-bit types.

Implementation Advice: If C interfacing is supported, the interface correspondences between Ada and C should be supported. 71.a/2

Implementation Advice: If the C implementation supports unsigned long long and long long, unsigned_long_long and long_long should be supported. 71.a.1/5

NOTE 1 Values of type char_array are not implicitly terminated with nul. If a char_array is to be passed as a parameter to an imported C function requiring nul termination, it is the programmer's responsibility to obtain this effect. 72

NOTE 2 To obtain the effect of C's sizeof(item_type), where Item_Type is the corresponding Ada type, evaluate the expression: size_t(Item_Type'Size/CHAR_BIT). 73

This paragraph was deleted.{*AI95-00216-01*} 74/2

NOTE 3 {*AI12-0028-1*} A variadic C function can correspond to several Ada subprograms, taking various specific numbers and types of parameters. 75/4

Examples

Example of using the Interfaces.C package: 76

```
{AI12-0312-1} --Calling the C Library Functions strcpy and printf                    77/5
with Interfaces.C;
procedure Test is
   package C renames Interfaces.C;
   use type C.char_array;
   -- Call <string.h>strcpy:
   -- C definition of strcpy:  char *strcpy(char *s1, const char *s2);
   --    This function copies the string pointed to by s2 (including the terminating null character)
   --    into the array pointed to by s1. If copying takes place between objects that overlap,
   --    the behavior is undefined. The strcpy function returns the value of s1.

{AI05-0229-1}      -- Note: since the C function's return value is of no interest, the Ada interface is a procedure   78/3
   procedure Strcpy (Target : out C.char_array;
                     Source : in  C.char_array)
      with Import => True, Convention => C, External_Name => "strcpy";

{AI05-0229-1} {AI12-0312-1}      -- Call <sdtio.h>printf:                          79/5
   -- C definition of printf:  int printf ( const char * format, ... );
   --    This function writes the C string pointed by format to the standard output (stdout).
   --    If format includes format specifiers (subsequences beginning with %), the additional
   --    arguments following format are formatted and inserted in the resulting string
   --    replacing their respective specifiers. If the number of arguments does not match
   --    the number of format specifiers, or if the types of the arguments do not match
   --    the corresponding format specifier, the behaviour is undefined. On success, the
   --    printf function returns the total number of characters written to the standard output.
   --    If a writing error occurs, a negative number is returned.

{AI12-0312-1}      -- Note: since the C function's return value is of no interest, the Ada interface is a procedure   79.1/5
   procedure Printf (Format : in C.char_array;
                     Param1 : in C.char_array;
                     Param2 : in C.int)
      with Import => True, Convention => C_Variadic_1, External_Name => "printf";

   Chars1 :  C.char_array(1..20);                                                    80
   Chars2 :  C.char_array(1..20);

begin                                                                                81
   Chars2(1..6) := "qwert" & C.nul;

   Strcpy(Chars1, Chars2);                                                           82

   -- Now Chars1(1..6) = "qwert" & C.Nul                                            83

{AI12-0312-1}     Printf("The String=%s, Length=%d", Chars1, Chars1'Length);       83.1/5

end Test;                                                                            84
```

Incompatibilities With Ada 95

{*AI95-00285-01*} {*AI05-0005-1*} Types char16_t and char32_t and their related types and operations are added to Interfaces.C. If Interfaces.C is referenced in a use_clause, and an entity *E* with the same defining_identifier as a new entity in Interfaces.C is defined in a package that is also referenced in a use_clause, the entity *E* may no longer be use-visible, resulting in errors. This should be rare and is easily fixed if it does occur. 84.a/3

Extensions to Ada 95

84.b/2 {*8652/0059*} {*AI95-00131-01*} **Corrigendum:** Convention C_Pass_By_Copy is new.

Wording Changes from Ada 95

84.c/2 {*8652/0060*} {*AI95-00037-01*} **Corrigendum:** Clarified the intent for Nul and Wide_Nul.

84.d/2 {*AI95-00216-01*} Specified that an unchecked union type (see B.3.3) is eligible for convention C_Pass_By_Copy.

84.e/2 {*AI95-00258-01*} Specified what happens if the To_C function tries to return a null string.

84.f/2 {*AI95-00337-01*} Clarified that the interface correspondences also apply to private types whose full types have the specified characteristics.

84.g/2 {*AI95-00343-01*} Clarified that a type must have convention C_Pass_By_Copy in order to be passed by copy (not just a type that could have that convention).

84.h/2 {*AI95-00376-01*} Added wording to make it clear that these facilities can also be used with C++.

Incompatibilities With Ada 2005

84.i/3 {*AI05-0002-1*} **Correction:** Added a definition of correspondences for function results. Also added wording to make it clear that we do not expect the implementation to conjure bounds for unconstrained arrays out of thin air. These changes allow (but don't require) compilers to reject unreasonable uses of array types. Such uses probably didn't work anyway (and probably were rejected, no matter what the language definition said), so little existing code should be impacted.

Incompatibilities With Ada 2012

84.j/5 {*AI12-0411-1*} Added type C_bool and (implicitly) the enumeration literals True and False to the Interfaces.C package. Therefore, a use clause conflict is possible; see the introduction of Annex A for more on this topic.

Extensions to Ada 2012

84.k/4 {*AI12-0028-1*} **Corrigendum:** The *convention*_identifiers C_Variadic_0, C_Variadic_1, and so on are new. These are classified as a correction as any implementation can add such identifiers and it is important that special conventions be available for variadic functions as typical x64 conventions are different for normal and variadic C functions.

84.l/4 {*AI12-0135-1*} **Corrigendum:** Defined the correspondence between an Ada enumeration type and a C enumeration type; implementations should support convention C for enumeration types.

Wording Changes from Ada 2012

84.m/5 {*AI12-0184-1*} Added Implementation Advice that types be defined in Interfaces.C corresponding to long long and unsigned long long.

84.n/5 {*AI12-0219-1*} **Correction:** Added advice that const t* map to Ada **in** parameters and vice versa.

84.o/5 {*AI12-0411-1*} Added advice about mapping type Boolean.

B.3.1 The Package Interfaces.C.Strings

1/3 {*AI05-0229-1*} The package Interfaces.C.Strings declares types and subprograms allowing an Ada program to allocate, reference, update, and free C-style strings. In particular, the private type chars_ptr corresponds to a common use of “char *” in C programs, and an object of this type can be passed to a subprogram to which **with** `Import => True, Convention => C` has been specified, and for which “char *” is the type of the argument of the C function.

Static Semantics

2 The library package Interfaces.C.Strings has the following declaration:

```
3/5     {AI12-0241-1} {AI12-0302-1} package Interfaces.C.Strings
           with Preelaborate, Nonblocking, Global => in out synchronized is

4          type char_array_access is access all char_array;

5/5     {AI95-00161-01} {AI12-0399-1}    type chars_ptr is private
              with Preelaborable_Initialization;
```

```
{AI95-00276-01}    type chars_ptr_array is array (size_t range <>) of aliased chars_ptr;
```
6/2

```
   Null_Ptr : constant chars_ptr;
```
7

```
   function To_Chars_Ptr (Item      : in char_array_access;
                          Nul_Check : in Boolean := False)
      return chars_ptr;
```
8

```
   function New_Char_Array (Chars   : in char_array) return chars_ptr;
```
9

```
   function New_String (Str : in String) return chars_ptr;
```
10

```
   procedure Free (Item : in out chars_ptr);
```
11

```
   Dereference_Error : exception;
```
12

```
   function Value (Item : in chars_ptr) return char_array;
```
13

```
   function Value (Item : in chars_ptr; Length : in size_t)
      return char_array;
```
14

```
   function Value (Item : in chars_ptr) return String;
```
15

```
   function Value (Item : in chars_ptr; Length : in size_t)
      return String;
```
16

```
   function Strlen (Item : in chars_ptr) return size_t;
```
17

```
   procedure Update (Item   : in chars_ptr;
                     Offset : in size_t;
                     Chars  : in char_array;
                     Check  : in Boolean := True);
```
18

```
   procedure Update (Item   : in chars_ptr;
                     Offset : in size_t;
                     Str    : in String;
                     Check  : in Boolean := True);
```
19

```
   Update_Error : exception;
```
20

```
private
```
21

```
   ... -- not specified by the language
end Interfaces.C.Strings;
```

Discussion: The string manipulation types and subprograms appear in a child of Interfaces.C versus being there directly, since it is useful to have Interfaces.C specified as pragma Pure. 21.a

Differently named functions New_String and New_Char_Array are declared, since if there were a single overloaded function a call with a string literal as actual parameter would be ambiguous. 21.b

The type chars_ptr is C-compatible and corresponds to the use of C's “char *” for a pointer to the first char in a char array terminated by nul. When an object of type chars_ptr is declared, its value is by default set to Null_Ptr, unless the object is imported (see B.1). 22

Discussion: The type char_array_access is not necessarily C-compatible, since an object of this type may carry “dope” information. The programmer should convert from char_array_access to chars_ptr for objects imported from, exported to, or passed to C. 22.a

```
function To_Chars_Ptr (Item      : in char_array_access;
                       Nul_Check : in Boolean := False)
   return chars_ptr;
```
23

{8652/0061} *{AI95-00140-01}* *{AI05-0264-1}* If Item is **null**, then To_Chars_Ptr returns Null_Ptr. If Item is not **null**, Nul_Check is True, and Item.**all** does not contain nul, then the function propagates Terminator_Error; otherwise, To_Chars_Ptr performs a pointer conversion with no allocation of memory. 24/3

```
function New_Char_Array (Chars   : in char_array) return chars_ptr;
```
25

This function returns a pointer to an allocated object initialized to Chars(Chars'First .. Index) & nul, where 26

- Index = Chars'Last if Chars does not contain nul, or 27

- Index is the smallest size_t value I such that Chars(I+1) = nul.

Storage_Error is propagated if the allocation fails.

```
function New_String (Str : in String) return chars_ptr;
```

This function is equivalent to New_Char_Array(To_C(Str)).

```
procedure Free (Item : in out chars_ptr);
```

If Item is Null_Ptr, then Free has no effect. Otherwise, Free releases the storage occupied by Value(Item), and resets Item to Null_Ptr.

```
function Value (Item : in chars_ptr) return char_array;
```

{*AI05-0264-1*} If Item = Null_Ptr, then Value propagates Dereference_Error. Otherwise, Value returns the prefix of the array of chars pointed to by Item, up to and including the first nul. The lower bound of the result is 0. If Item does not point to a nul-terminated string, then execution of Value is erroneous.

```
function Value (Item : in chars_ptr; Length : in size_t)
   return char_array;
```

{*8652/0062*} {*AI95-00139-01*} {*AI05-0264-1*} If Item = Null_Ptr, then Value propagates Dereference_Error. Otherwise, Value returns the shorter of two arrays, either the first Length chars pointed to by Item, or Value(Item). The lower bound of the result is 0. If Length is 0, then Value propagates Constraint_Error.

Ramification: Value(New_Char_Array(Chars)) = Chars if Chars does not contain nul; else Value(New_Char_Array(Chars)) is the prefix of Chars up to and including the first nul.

```
function Value (Item : in chars_ptr) return String;
```

Equivalent to To_Ada(Value(Item), Trim_Nul=>True).

```
function Value (Item : in chars_ptr; Length : in size_t)
   return String;
```

{*8652/0063*} {*AI95-00177-01*} Equivalent to To_Ada(Value(Item, Length) & nul, Trim_Nul=>True).

```
function Strlen (Item : in chars_ptr) return size_t;
```

Returns *Val*'Length–1 where *Val* = Value(Item); propagates Dereference_Error if Item = Null_Ptr.

Ramification: Strlen returns the number of chars in the array pointed to by Item, up to and including the char immediately before the first nul.

Strlen has the same possibility for erroneous execution as Value, in cases where the string has not been nul-terminated.

Strlen has the effect of C's strlen function.

```
procedure Update (Item   : in chars_ptr;
                  Offset : in size_t;
                  Chars  : in char_array;
                  Check  : Boolean := True);
```

{*8652/0064*} {*AI95-00039-01*} If Item = Null_Ptr, then Update propagates Dereference_Error. Otherwise, this procedure updates the value pointed to by Item, starting at position Offset, using Chars as the data to be copied into the array. Overwriting the nul terminator, and skipping with the Offset past the nul terminator, are both prevented if Check is True, as follows:

- Let N = Strlen(Item). If Check is True, then:
  - If Offset+Chars'Length>N, propagate Update_Error.

- Otherwise, overwrite the data in the array pointed to by Item, starting at the char at position Offset, with the data in Chars. 47

- If Check is False, then processing is as above, but with no check that Offset+Chars'Length>N. 48

Ramification: If Chars contains nul, Update's effect may be to “shorten” the pointed-to char array. 48.a

```
procedure Update (Item   : in chars_ptr;                                    49
                  Offset : in size_t;
                  Str    : in String;
                  Check  : in Boolean := True);
```

{*AI95-00242-01*} Equivalent to Update(Item, Offset, To_C(Str, Append_Nul => False), Check). 50/2

Discussion: {*AI95-00242-01*} To truncate the Item to the length of Str, use Update(Item, Offset, To_C(Str), Check) instead of Update(Item, Offset, Str, Check). Note that when truncating Item, Item must be longer than Str. 50.a/2

Erroneous Execution

Execution of any of the following is erroneous if the Item parameter is not null_ptr and Item does not point to a nul-terminated array of chars. 51

- a Value function not taking a Length parameter, 52
- the Free procedure, 53
- the Strlen function. 54

Execution of Free(X) is also erroneous if the chars_ptr X was not returned by New_Char_Array or New_String. 55

Reading or updating a freed char_array is erroneous. 56

Execution of Update is erroneous if Check is False and a call with Check equal to True would have propagated Update_Error. 57

NOTE {*AI12-0440-1*} New_Char_Array and New_String can be implemented either through the allocation function from the C environment (“malloc”) or through Ada dynamic memory allocation (“new”). The key points are 58/5

- {*AI12-0440-1*} the returned value (a chars_ptr) is represented as a C “char *” so that it can be passed to C functions; 59/5
- {*AI12-0442-1*} the allocated object can be freed by the programmer via a call of Free, rather than by calling a C function. 60/5

Inconsistencies With Ada 95

{*AI95-00242-01*} **Amendment Correction:** Update for a String parameter is now defined to not add a nul character. It did add a nul in Ada 95. This means that programs that used this behavior of Update to truncate a string will no longer work (the string will not be truncated). This change makes Update for a string consistent with Update for a char_array (no implicit nul is added to the end of a char_array). 60.a/2

Extensions to Ada 95

{*AI95-00161-01*} **Amendment Correction:** Added pragma Preelaborable_Initialization to type chars_ptr, so that it can be used in preelaborated units. 60.b/2

{*AI95-00276-01*} **Amendment Correction:** The components of chars_ptr_array are aliased so that it can be used to instantiate Interfaces.C.Pointers (that is its intended purpose, which is otherwise mysterious as it has no operations). 60.c/2

Wording Changes from Ada 95

{*8652/0061*} {*AI95-00140-01*} **Corrigendum:** Fixed the missing semantics of To_Char_Ptr when Nul_Check is False. 60.d/2

{*8652/0062*} {*AI95-00139-01*} **Corrigendum:** Fixed the missing semantics of Value when the Length is 0. 60.e/2

{*8652/0063*} {*AI95-00177-01*} **Corrigendum:** Corrected the definition of Value to avoid raising Terminator_Error. 60.f/2

{*8652/0064*} {*AI95-00039-01*} **Corrigendum:** Fixed the missing semantics of Update when Item is Null_Ptr. 60.g/2

B.3.2 The Generic Package Interfaces.C.Pointers

1 The generic package Interfaces.C.Pointers allows the Ada programmer to perform C-style operations on pointers. It includes an access type Pointer, Value functions that dereference a Pointer and deliver the designated array, several pointer arithmetic operations, and “copy” procedures that copy the contents of a source pointer into the array designated by a destination pointer. As in C, it treats an object Ptr of type Pointer as a pointer to the first element of an array, so that for example, adding 1 to Ptr yields a pointer to the second element of the array.

2/5 {*AI12-0445-1*} The generic allows two styles of usage: one in which the array is terminated by a special terminator element; and another in which the programmer keeps track of the length.

Static Semantics

3 The generic library package Interfaces.C.Pointers has the following declaration:

4/5
```
{AI12-0241-1} {AI12-0302-1} generic
   type Index is (<>);
   type Element is private;
   type Element_Array is array (Index range <>) of aliased Element;
   Default_Terminator : Element;
package Interfaces.C.Pointers
   with Preelaborate, Nonblocking, Global => in out synchronized is
```

5
```
   type Pointer is access all Element;
```

6
```
   function Value(Ref        : in Pointer;
                  Terminator : in Element := Default_Terminator)
      return Element_Array;
```

7
```
   function Value(Ref    : in Pointer;
                  Length : in ptrdiff_t)
      return Element_Array;
```

8
```
   Pointer_Error : exception;
```

9
```
   -- C-style Pointer arithmetic
```

10/3
```
   {AI05-0229-1}    function "+" (Left : in Pointer;   Right : in ptrdiff_t) return
   Pointer
         with Convention => Intrinsic;
      function "+" (Left : in ptrdiff_t; Right : in Pointer)   return Pointer
         with Convention => Intrinsic;
      function "-" (Left : in Pointer;   Right : in ptrdiff_t) return Pointer
         with Convention => Intrinsic;
      function "-" (Left : in Pointer;   Right : in Pointer) return ptrdiff_t
         with Convention => Intrinsic;
```

11/3
```
   {AI05-0229-1}    procedure Increment (Ref : in out Pointer)
         with Convention => Intrinsic;
      procedure Decrement (Ref : in out Pointer)
         with Convention => Intrinsic;
```

12/3 *This paragraph was deleted.*{*AI05-0229-1*}

13
```
   function Virtual_Length (Ref        : in Pointer;
                            Terminator : in Element := Default_Terminator)
      return ptrdiff_t;
```

14
```
   procedure Copy_Terminated_Array
      (Source     : in Pointer;
       Target     : in Pointer;
       Limit      : in ptrdiff_t := ptrdiff_t'Last;
       Terminator : in Element :=  Default_Terminator);
```

15
```
   procedure Copy_Array (Source  : in Pointer;
                         Target  : in Pointer;
                         Length  : in ptrdiff_t);
```

16
```
end Interfaces.C.Pointers;
```

The type Pointer is C-compatible and corresponds to one use of C's “Element *”. An object of type Pointer is interpreted as a pointer to the initial Element in an Element_Array. Two styles are supported: 17

- Explicit termination of an array value with Default_Terminator (a special terminator value); 18
- Programmer-managed length, with Default_Terminator treated simply as a data element. 19

```
function Value(Ref        : in Pointer;
               Terminator : in Element := Default_Terminator)
   return Element_Array;
```
20

This function returns an Element_Array whose value is the array pointed to by Ref, up to and including the first Terminator; the lower bound of the array is Index'First. Interfaces.C.Strings.Dereference_Error is propagated if Ref is **null**. 21

```
function Value(Ref    : in Pointer;
               Length : in ptrdiff_t)
   return Element_Array;
```
22

This function returns an Element_Array comprising the first Length elements pointed to by Ref. The exception Interfaces.C.Strings.Dereference_Error is propagated if Ref is **null**. 23

The "+" and "–" functions perform arithmetic on Pointer values, based on the Size of the array elements. In each of these functions, Pointer_Error is propagated if a Pointer parameter is **null**. 24

```
procedure Increment (Ref : in out Pointer);
```
25

Equivalent to Ref := Ref+1. 26

```
procedure Decrement (Ref : in out Pointer);
```
27

Equivalent to Ref := Ref–1. 28

```
function Virtual_Length (Ref        : in Pointer;
                         Terminator : in Element := Default_Terminator)
   return ptrdiff_t;
```
29

Returns the number of Elements, up to the one just before the first Terminator, in Value(Ref, Terminator). 30

```
procedure Copy_Terminated_Array
   (Source     : in Pointer;
    Target     : in Pointer;
    Limit      : in ptrdiff_t := ptrdiff_t'Last;
    Terminator : in Element := Default_Terminator);
```
31

This procedure copies Value(Source, Terminator) into the array pointed to by Target; it stops either after Terminator has been copied, or the number of elements copied is Limit, whichever occurs first. Dereference_Error is propagated if either Source or Target is **null**. 32

Ramification: It is the programmer's responsibility to ensure that elements are not copied beyond the logical length of the target array. 32.a

Implementation Note: The implementation has to take care to check the Limit first. 32.b

```
procedure Copy_Array (Source  : in Pointer;
                      Target  : in Pointer;
                      Length  : in ptrdiff_t);
```
33

This procedure copies the first Length elements from the array pointed to by Source, into the array pointed to by Target. Dereference_Error is propagated if either Source or Target is **null**. 34

Erroneous Execution

It is erroneous to dereference a Pointer that does not designate an aliased Element. 35

35.a **Discussion:** Such a Pointer could arise via "+", "–", Increment, or Decrement.

36 Execution of Value(Ref, Terminator) is erroneous if Ref does not designate an aliased Element in an Element_Array terminated by Terminator.

37 Execution of Value(Ref, Length) is erroneous if Ref does not designate an aliased Element in an Element_Array containing at least Length Elements between the designated Element and the end of the array, inclusive.

38 Execution of Virtual_Length(Ref, Terminator) is erroneous if Ref does not designate an aliased Element in an Element_Array terminated by Terminator.

39 Execution of Copy_Terminated_Array(Source, Target, Limit, Terminator) is erroneous in either of the following situations:

40 • Execution of both Value(Source, Terminator) and Value(Source, Limit) are erroneous, or

41 • Copying writes past the end of the array containing the Element designated by Target.

42 Execution of Copy_Array(Source, Target, Length) is erroneous if either Value(Source, Length) is erroneous, or copying writes past the end of the array containing the Element designated by Target.

43 NOTE To compose a Pointer from an Element_Array, use 'Access on the first element. For example (assuming appropriate instantiations):

44
```
Some_Array   : Element_Array(0..5) ;
Some_Pointer : Pointer := Some_Array(0)'Access;
```

Examples

45 *Example of Interfaces.C.Pointers:*

46
```
with Interfaces.C.Pointers;
with Interfaces.C.Strings;
procedure Test_Pointers is
   package C renames Interfaces.C;
   package Char_Ptrs is
      new C.Pointers (Index              => C.size_t,
                      Element            => C.char,
                      Element_Array      => C.char_array,
                      Default_Terminator => C.nul);
```

47
```
   use type Char_Ptrs.Pointer;
   subtype Char_Star is Char_Ptrs.Pointer;
```

48
```
   procedure Strcpy (Target_Ptr, Source_Ptr : Char_Star) is
      Target_Temp_Ptr : Char_Star := Target_Ptr;
      Source_Temp_Ptr : Char_Star := Source_Ptr;
      Element : C.char;
   begin
      if Target_Temp_Ptr = null or Source_Temp_Ptr = null then
         raise C.Strings.Dereference_Error;
      end if;
```

49/1
```
{8652/0065} {AI95-00142-01}       loop
         Element             := Source_Temp_Ptr.all;
         Target_Temp_Ptr.all := Element;
         exit when C."="(Element, C.nul);
         Char_Ptrs.Increment(Target_Temp_Ptr);
         Char_Ptrs.Increment(Source_Temp_Ptr);
      end loop;
   end Strcpy;
begin
   ...
end Test_Pointers;
```

B.3.3 Unchecked Union Types

{*AI95-00216-01*} {*AI05-0229-1*} {*AI05-0269-1*} [Specifying aspect Unchecked_Union to have the value True defines an interface correspondence between a given discriminated type and some C union. The aspect requires that the associated type shall be given a representation that allocates no space for its discriminant(s).] 1/3

Paragraphs 2 through 3 were moved to Annex J, "Obsolescent Features".

Static Semantics

{*AI05-0229-1*} For a discriminated record type having a variant_part, the following language-defined representation aspect may be specified: 3.1/3

Unchecked_Union 3.2/3

The type of aspect Unchecked_Union is Boolean. If directly specified, the aspect_definition shall be a static expression. If not specified (including by inheritance), the aspect is False.

Aspect Description for Unchecked_Union: Type is used to interface to a C union type. 3.a/3

Legality Rules

Paragraphs 4 and 5 were deleted.

{*AI95-00216-01*} {*AI05-0229-1*} A type for which aspect Unchecked_Union is True is called an *unchecked union type*. A subtype of an unchecked union type is defined to be an *unchecked union subtype*. An object of an unchecked union type is defined to be an *unchecked union object*. 6/3

{*AI95-00216-01*} All component subtypes of an unchecked union type shall be C-compatible. 7/2

{*AI95-00216-01*} If a component subtype of an unchecked union type is subject to a per-object constraint, then the component subtype shall be an unchecked union subtype. 8/2

{*AI95-00216-01*} {*AI05-0026-1*} {*AI12-0174-1*} Any name that denotes a discriminant of an object of an unchecked union type shall occur within the declarative region of the type or as the selector_name of an aggregate, and shall not occur within a record_representation_clause. 9/5

{*AI95-00216-01*} {*AI05-0026-1*} The type of a component declared in a variant_part of an unchecked union type shall not need finalization. In addition to the places where Legality Rules normally apply (see 12.3), this rule also applies in the private part of an instance of a generic unit. For an unchecked union type declared within the body of a generic unit, or within the body of any of its descendant library units, no part of the type of a component declared in a variant_part of the unchecked union type shall be of a formal private type or formal private extension declared within the formal part of the generic unit. 10/3

Reason: {*AI05-0026-1*} The last part is a classic assume-the-worst rule that avoids dependence on the actuals in a generic body. We did not include this in the definition of "needs finalization" as it has a bad interaction with the use of that term for the No_Nested_Finalization restriction. 10.a/3

{*AI95-00216-01*} The completion of an incomplete or private type declaration having a known_discriminant_part shall not be an unchecked union type. 11/2

{*AI95-00216-01*} An unchecked union subtype shall only be passed as a generic actual parameter if the corresponding formal type has no known discriminants or is an unchecked union type. 12/2

Ramification: This includes formal private types without a known_discriminant_part, formal derived types that do not inherit any discriminants (formal derived types do not have known_discriminant_parts), and formal derived types that are unchecked union types. 12.a/2

Static Semantics

13/2 {*AI95-00216-01*} An unchecked union type is eligible for convention C.

14/2 {*AI95-00216-01*} All objects of an unchecked union type have the same size.

15/2 {*AI95-00216-01*} Discriminants of objects of an unchecked union type are of size zero.

16/2 {*AI95-00216-01*} Any check which would require reading a discriminant of an unchecked union object is suppressed (see 11.5). These checks include:

17/5 - {*AI12-0453-1*} The check performed when addressing a variant component (that is, a component that was declared in a variant part) of an unchecked union object that the object has this component (see 4.1.3).

18/2 - Any checks associated with a type or subtype conversion of a value of an unchecked union type (see 4.6). This includes, for example, the check associated with the implicit subtype conversion of an assignment statement.

19/2 - The subtype membership check associated with the evaluation of a qualified expression (see 4.7) or an uninitialized allocator (see 4.8).

19.a/2 **Discussion:** If a suppressed check would have failed, execution is erroneous (see 11.5). An implementation is always allowed to make a suppressed check if it can somehow determine the discriminant value.

Dynamic Semantics

20/2 {*AI95-00216-01*} A view of an unchecked union object (including a type conversion or function call) has *inferable discriminants* if it has a constrained nominal subtype, unless the object is a component of an enclosing unchecked union object that is subject to a per-object constraint and the enclosing object lacks inferable discriminants.

21/2 {*AI95-00216-01*} An expression of an unchecked union type has inferable discriminants if it is either a name of an object with inferable discriminants or a qualified expression whose subtype_mark denotes a constrained subtype.

22/2 {*AI95-00216-01*} Program_Error is raised in the following cases:

23/2 - Evaluation of the predefined equality operator for an unchecked union type if either of the operands lacks inferable discriminants.

24/2 - Evaluation of the predefined equality operator for a type which has a subcomponent of an unchecked union type whose nominal subtype is unconstrained.

25/5 - {*AI12-0162-1*} Evaluation of an individual membership test if the subtype_mark (if any) denotes a constrained unchecked union subtype and the *tested*_simple_expression lacks inferable discriminants.

26/2 - Conversion from a derived unchecked union type to an unconstrained non-unchecked-union type if the operand of the conversion lacks inferable discriminants.

27/2 - Execution of the default implementation of the Write or Read attribute of an unchecked union type.

28/2 - Execution of the default implementation of the Output or Input attribute of an unchecked union type if the type lacks default discriminant values.

Paragraph 29 was deleted.

NOTE {*AI95-00216-01*} The use of an unchecked union to obtain the effect of an unchecked conversion results in erroneous execution (see 11.5). Execution of the following example is erroneous even if Float'Size = Integer'Size: 30/2

```
{AI05-0229-1} type T (Flag : Boolean := False) is                                    31/3
   record
       case Flag is
           when False =>
               F1 : Float := 0.0;
           when True =>
               F2 : Integer := 0;
       end case;
    end record
    with Unchecked_Union;
X : T;                                                                               32/2
Y : Integer := X.F2; -- erroneous
```

Extensions to Ada 95

{*AI95-00216-01*} Pragma Unchecked_Union is new. 32.a/2

Incompatibilities With Ada 2005

{*AI05-0026-1*} **Correction:** The use of discriminants on Unchecked_Union types is now illegal in record_representation_clauses, as it makes no sense to specify a position for something that is not supposed to exist. It is very unlikely that this change will have any impact on existing code. 32.b/3

Extensions to Ada 2005

{*AI05-0229-1*} Aspect Unchecked_Union is new; pragma Unchecked_Union is now obsolescent. 32.c/3

Wording Changes from Ada 2005

{*AI05-0026-1*} **Correction:** Revised the rules to use the "needs finalization" definition, and eliminated generic contract issues. 32.d/3

Wording Changes from Ada 2012

{*AI05-0162-1*} **Correction:** Adjusted the wording to reflect that membership tests can have more than one expression or subtype_mark. 32.e/5

{*AI05-0174-1*} **Correction:** Adjusted the wording to allow named aggregates of an unchecked union type; it is clearly madness to allow positional record components in an aggregate but not named component associations. 32.f/5

B.4 Interfacing with COBOL

{*AI05-0229-1*} The facilities relevant to interfacing with the COBOL language are the package Interfaces.COBOL and support for specifying the Convention aspect with *convention*_identifier COBOL. 1/3

The COBOL interface package supplies several sets of facilities: 2

- A set of types corresponding to the native COBOL types of the supported COBOL implementation (so-called "internal COBOL representations"), allowing Ada data to be passed as parameters to COBOL programs 3
- {*AI12-0439-1*} A set of types and constants reflecting external data representations such as can be found in files or databases, allowing COBOL-generated data to be read by an Ada program, and Ada-generated data to be read by COBOL programs 4/5
- A generic package for converting between an Ada decimal type value and either an internal or external COBOL representation 5

Static Semantics

The library package Interfaces.COBOL has the following declaration: 6

```
{AI12-0241-1} {AI12-0302-1} package Interfaces.COBOL                                 7/5
   with Preelaborate, Nonblocking, Global => in out synchronized is
   -- Types and operations for internal data representations                         8
```

```
9          type Floating      is digits implementation-defined;
           type Long_Floating is digits implementation-defined;

10         type Binary      is range implementation-defined;
           type Long_Binary is range implementation-defined;

11         Max_Digits_Binary      : constant := implementation-defined;
           Max_Digits_Long_Binary : constant := implementation-defined;

12/3    {AI05-0229-1}    type Decimal_Element  is mod implementation-defined;
           type Packed_Decimal is array (Positive range <>) of Decimal_Element
              with Pack;

13         type COBOL_Character is implementation-defined character type;

14         Ada_To_COBOL : array (Character) of COBOL_Character := implementation-defined;

15         COBOL_To_Ada : array (COBOL_Character) of Character := implementation-defined;

16/3    {AI05-0229-1}    type Alphanumeric is array (Positive range <>) of COBOL_Character
              with Pack;

17         function To_COBOL (Item : in String) return Alphanumeric;
           function To_Ada   (Item : in Alphanumeric) return String;

18         procedure To_COBOL (Item       : in String;
                               Target     : out Alphanumeric;
                               Last       : out Natural);

19         procedure To_Ada (Item     : in Alphanumeric;
                             Target   : out String;
                             Last     : out Natural);

20/3    {AI05-0229-1}    type Numeric is array (Positive range <>) of COBOL_Character
              with Pack;

21      -- Formats for COBOL data representations

22         type Display_Format is private;

23         Unsigned             : constant Display_Format;
           Leading_Separate     : constant Display_Format;
           Trailing_Separate    : constant Display_Format;
           Leading_Nonseparate  : constant Display_Format;
           Trailing_Nonseparate : constant Display_Format;

24         type Binary_Format is private;

25         High_Order_First  : constant Binary_Format;
           Low_Order_First   : constant Binary_Format;
           Native_Binary     : constant Binary_Format;

26         type Packed_Format is private;

27         Packed_Unsigned   : constant Packed_Format;
           Packed_Signed     : constant Packed_Format;

28      -- Types for external representation of COBOL binary data

29/3    {AI05-0229-1}    type Byte is mod 2**COBOL_Character'Size;
           type Byte_Array is array (Positive range <>) of Byte
              with Pack;

30         Conversion_Error : exception;

31         generic
              type Num is delta <> digits <>;
           package Decimal_Conversions is

32            -- Display Formats: data values are represented as Numeric

33            function Valid (Item   : in Numeric;
                              Format : in Display_Format) return Boolean;

34            function Length (Format : in Display_Format) return Natural;

35            function To_Decimal (Item   : in Numeric;
                                   Format : in Display_Format) return Num;

36            function To_Display (Item   : in Num;
                                   Format : in Display_Format) return Numeric;
```

```
      -- Packed Formats: data values are represented as Packed_Decimal                    37
      function Valid (Item   : in Packed_Decimal;                                        38
                      Format : in Packed_Format) return Boolean;
      function Length (Format : in Packed_Format) return Natural;                        39
      function To_Decimal (Item   : in Packed_Decimal;                                   40
                           Format : in Packed_Format) return Num;
      function To_Packed (Item   : in Num;                                               41
                          Format : in Packed_Format) return Packed_Decimal;
      -- Binary Formats: external data values are represented as Byte_Array              42
      function Valid (Item   : in Byte_Array;                                            43
                      Format : in Binary_Format) return Boolean;
      function Length (Format : in Binary_Format) return Natural;                        44
      function To_Decimal (Item   : in Byte_Array;
                           Format : in Binary_Format) return Num;
      function To_Binary (Item   : in Num;                                               45
                          Format : in Binary_Format) return Byte_Array;
      -- Internal Binary formats: data values are of type Binary or Long_Binary          46
      function To_Decimal (Item : in Binary)      return Num;                            47
      function To_Decimal (Item : in Long_Binary) return Num;
      function To_Binary      (Item : in Num)  return Binary;                            48
      function To_Long_Binary (Item : in Num)  return Long_Binary;
   end Decimal_Conversions;                                                              49
private                                                                                  50
   ... -- not specified by the language
end Interfaces.COBOL;
```

Implementation defined: The types Floating, Long_Floating, Binary, Long_Binary, Decimal_Element, and COBOL_Character; and the initializations of the variables Ada_To_COBOL and COBOL_To_Ada, in Interfaces.COBOL. 50.a/1

Each of the types in Interfaces.COBOL is COBOL-compatible. 51

The types Floating and Long_Floating correspond to the native types in COBOL for data items with computational usage implemented by floating point. The types Binary and Long_Binary correspond to the native types in COBOL for data items with binary usage, or with computational usage implemented by binary. 52

Max_Digits_Binary is the largest number of decimal digits in a numeric value that is represented as Binary. Max_Digits_Long_Binary is the largest number of decimal digits in a numeric value that is represented as Long_Binary. 53

The type Packed_Decimal corresponds to COBOL's packed-decimal usage. 54

The type COBOL_Character defines the run-time character set used in the COBOL implementation. Ada_To_COBOL and COBOL_To_Ada are the mappings between the Ada and COBOL run-time character sets. 55

Reason: The character mappings are visible variables, since the user needs the ability to modify them at run time. 55.a

Type Alphanumeric corresponds to COBOL's alphanumeric data category. 56

Each of the functions To_COBOL and To_Ada converts its parameter based on the mappings Ada_To_COBOL and COBOL_To_Ada, respectively. The length of the result for each is the length of the parameter, and the lower bound of the result is 1. Each component of the result is obtained by applying the relevant mapping to the corresponding component of the parameter. 57

Each of the procedures To_COBOL and To_Ada copies converted elements from Item to Target, using the appropriate mapping (Ada_To_COBOL or COBOL_To_Ada, respectively). The index in Target of 58

the last element assigned is returned in Last (0 if Item is a null array). If Item'Length exceeds Target'Length, Constraint_Error is propagated.

59 Type Numeric corresponds to COBOL's numeric data category with display usage.

60 The types Display_Format, Binary_Format, and Packed_Format are used in conversions between Ada decimal type values and COBOL internal or external data representations. The value of the constant Native_Binary is either High_Order_First or Low_Order_First, depending on the implementation.

61
```
function Valid (Item   : in Numeric;
                Format : in Display_Format) return Boolean;
```

62 The function Valid checks that the Item parameter has a value consistent with the value of Format. If the value of Format is other than Unsigned, Leading_Separate, and Trailing_Separate, the effect is implementation defined. If Format does have one of these values, the following rules apply:

63/3 - {*8652/0066*} {*AI95-00071-01*} {*AI05-0264-1*} Format=Unsigned: if Item comprises one or more decimal digit characters, then Valid returns True, else it returns False.

64/1 - {*8652/0066*} {*AI95-00071-01*} Format=Leading_Separate: if Item comprises a single occurrence of the plus or minus sign character, and then one or more decimal digit characters, then Valid returns True, else it returns False.

65/1 - {*8652/0066*} {*AI95-00071-01*} Format=Trailing_Separate: if Item comprises one or more decimal digit characters and finally a plus or minus sign character, then Valid returns True, else it returns False.

66
```
function Length (Format : in Display_Format) return Natural;
```

67 The Length function returns the minimal length of a Numeric value sufficient to hold any value of type Num when represented as Format.

68
```
function To_Decimal (Item   : in Numeric;
                     Format : in Display_Format) return Num;
```

69 Produces a value of type Num corresponding to Item as represented by Format. The number of digits after the assumed radix point in Item is Num'Scale. Conversion_Error is propagated if the value represented by Item is outside the range of Num.

69.a **Discussion:** There is no issue of truncation versus rounding, since the number of decimal places is established by Num'Scale.

70
```
function To_Display (Item   : in Num;
                     Format : in Display_Format) return Numeric;
```

71/1 {*8652/0067*} {*AI95-00072-01*} This function returns the Numeric value for Item, represented in accordance with Format. The length of the returned value is Length(Format), and the lower bound is 1. Conversion_Error is propagated if Num is negative and Format is Unsigned.

72
```
function Valid (Item   : in Packed_Decimal;
                Format : in Packed_Format) return Boolean;
```

73 This function returns True if Item has a value consistent with Format, and False otherwise. The rules for the formation of Packed_Decimal values are implementation defined.

74
```
function Length (Format : in Packed_Format) return Natural;
```

75 This function returns the minimal length of a Packed_Decimal value sufficient to hold any value of type Num when represented as Format.

```
function To_Decimal (Item   : in Packed_Decimal;                                    76
                     Format : in Packed_Format) return Num;
```

Produces a value of type Num corresponding to Item as represented by Format. Num'Scale is the number of digits after the assumed radix point in Item. Conversion_Error is propagated if the value represented by Item is outside the range of Num. 77

```
function To_Packed (Item   : in Num;                                                78
                    Format : in Packed_Format) return Packed_Decimal;
```

{*8652/0067*} {*AI95-00072-01*} This function returns the Packed_Decimal value for Item, represented in accordance with Format. The length of the returned value is Length(Format), and the lower bound is 1. Conversion_Error is propagated if Num is negative and Format is Packed_Unsigned. 79/1

```
function Valid (Item   : in Byte_Array;                                             80
                Format : in Binary_Format) return Boolean;
```

This function returns True if Item has a value consistent with Format, and False otherwise. 81

Ramification: This function returns False only when the represented value is outside the range of Num. 81.a

```
function Length (Format : in Binary_Format) return Natural;                         82
```

This function returns the minimal length of a Byte_Array value sufficient to hold any value of type Num when represented as Format. 83

```
function To_Decimal (Item   : in Byte_Array;                                        84
                     Format : in Binary_Format) return Num;
```

Produces a value of type Num corresponding to Item as represented by Format. Num'Scale is the number of digits after the assumed radix point in Item. Conversion_Error is propagated if the value represented by Item is outside the range of Num. 85

```
function To_Binary (Item   : in Num;                                                86
                    Format : in Binary_Format) return Byte_Array;
```

{*8652/0067*} {*AI95-00072-01*} This function returns the Byte_Array value for Item, represented in accordance with Format. The length of the returned value is Length(Format), and the lower bound is 1. 87/1

```
function To_Decimal (Item : in Binary)      return Num;                             88

function To_Decimal (Item : in Long_Binary) return Num;
```

These functions convert from COBOL binary format to a corresponding value of the decimal type Num. Conversion_Error is propagated if Item is too large for Num. 89

Ramification: There is no rescaling performed on the conversion. That is, the returned value in each case is a “bit copy” if Num has a binary radix. The programmer is responsible for maintaining the correct scale. 89.a

```
function To_Binary      (Item : in Num)  return Binary;                             90

function To_Long_Binary (Item : in Num)  return Long_Binary;
```

These functions convert from Ada decimal to COBOL binary format. Conversion_Error is propagated if the value of Item is too large to be represented in the result type. 91

Discussion: One style of interface supported for COBOL, similar to what is provided for C, is the ability to call and pass parameters to an existing COBOL program. Thus the interface package supplies types that can be used in an Ada program as parameters to subprograms whose bodies will be in COBOL. These types map to COBOL's alphanumeric and numeric data categories. 91.a

Several types are provided for support of alphanumeric data. Since COBOL's run-time character set is not necessarily the same as Ada's, Interfaces.COBOL declares an implementation-defined character type COBOL_Character, and mappings between Character and COBOL_Character. These mappings are visible variables (rather than, say, functions 91.b

or constant arrays), since in the situation where COBOL_Character is EBCDIC, the flexibility of dynamically modifying the mappings is needed. Corresponding to COBOL's alphanumeric data is the string type Alphanumeric.

91.c Numeric data may have either a “display” or “computational” representation in COBOL. On the Ada side, the data is of a decimal fixed point type. Passing an Ada decimal data item to a COBOL program requires conversion from the Ada decimal type to some type that reflects the representation expected on the COBOL side.

91.d • Computational Representation

91.e Floating point representation is modeled by Ada floating point types, Floating and Long_Floating. Conversion between these types and Ada decimal types is obtained directly, since the type name serves as a conversion function.

91.f Binary representation is modeled by an Ada integer type, Binary, and possibly other types such as Long_Binary. Conversion between, say, Binary and a decimal type is through functions from an instantiation of the generic package Decimal_Conversions.

91.g Packed decimal representation is modeled by the Ada array type Packed_Decimal. Conversion between packed decimal and a decimal type is through functions from an instantiation of the generic package Decimal_Conversions.

91.h • Display Representation

91.i Display representation for numeric data is modeled by the array type Numeric. Conversion between display representation and a decimal type is through functions from an instantiation of the generic package Decimal_Conversions. A parameter to the conversion function indicates the desired interpretation of the data (e.g., signed leading separate, etc.)

91.j/3 {*AI05-0229-1*} The Convention of a record type may be specified as COBOL to direct the compiler to choose a COBOL-compatible representation for objects of the type.

91.k The package Interfaces.COBOL allows the Ada programmer to deal with data from files (or databases) created by a COBOL program. For data that is alphanumeric, or in display or packed decimal format, the approach is the same as for passing parameters (instantiate Decimal_Conversions to obtain the needed conversion functions). For binary data, the external representation is treated as a Byte array, and an instantiation of Decimal_IO produces a package that declares the needed conversion functions. A parameter to the conversion function indicates the desired interpretation of the data (e.g., high- versus low-order byte first).

Implementation Requirements

92/3 {*AI05-0229-1*} An implementation shall support specifying aspect Convention with a COBOL *convention*_identifier for a COBOL-eligible type (see B.1).

92.a **Ramification:** An implementation supporting this package shall ensure that if the bounds of a Packed_Decimal, Alphanumeric, or Numeric variable are static, then the representation of the object comprises solely the array components (that is, there is no implicit run-time “descriptor” that is part of the object).

Implementation Permissions

93 An implementation may provide additional constants of the private types Display_Format, Binary_Format, or Packed_Format.

93.a **Reason:** This is to allow exploitation of other external formats that may be available in the COBOL implementation.

94 An implementation may provide further floating point and integer types in Interfaces.COBOL to match additional native COBOL types, and may also supply corresponding conversion functions in the generic package Decimal_Conversions.

Implementation Advice

95 An Ada implementation should support the following interface correspondences between Ada and COBOL.

96 • An Ada **access** T parameter is passed as a “BY REFERENCE” data item of the COBOL type corresponding to T.

97 • An Ada **in** scalar parameter is passed as a “BY CONTENT” data item of the corresponding COBOL type.

- Any other Ada parameter is passed as a “BY REFERENCE” data item of the COBOL type corresponding to the Ada parameter type; for scalars, a local copy is used if necessary to ensure by-copy semantics. 98

Implementation Advice: If COBOL interfacing is supported, the interface correspondences between Ada and COBOL should be supported. 98.a/2

NOTE 1 {*AI05-0229-1*} An implementation is not required to support specifying aspect Convention for access types, nor is it required to support specifying aspects Import, Export, or Convention for functions. 99/3

Reason: COBOL does not have a pointer facility, and a COBOL program does not return a value. 99.a

NOTE 2 {*AI12-0440-1*} If an Ada subprogram is exported to COBOL, then a call from COBOL call can specify either “BY CONTENT” or “BY REFERENCE”. 100/5

Examples

Examples of Interfaces.COBOL: 101

```
with Interfaces.COBOL;                                                          102
procedure Test_Call is

   -- Calling a foreign COBOL program                                           103
   -- Assume that a COBOL program PROG has the following declaration
   --  in its LINKAGE section:
   --  01 Parameter-Area
   --    05 NAME   PIC X(20).
   --    05 SSN    PIC X(9).
   --    05 SALARY PIC 99999V99 USAGE COMP.
   -- The effect of PROG is to update SALARY based on some algorithm

   package COBOL renames Interfaces.COBOL;                                      104

   type Salary_Type is delta 0.01 digits 7;                                     105

{AI05-0229-1}     type COBOL_Record is                                          106/3
      record
         Name   : COBOL.Numeric(1..20);
         SSN    : COBOL.Numeric(1..9);
         Salary : COBOL.Binary;  -- Assume Binary = 32 bits
      end record
      with Convention => COBOL;

{AI05-0229-1}     procedure Prog (Item : in out COBOL_Record)                   107/3
      with Import => True, Convention => COBOL;

   package Salary_Conversions is                                                108
      new COBOL.Decimal_Conversions(Salary_Type);

   Some_Salary : Salary_Type := 12_345.67;                                      109
   Some_Record : COBOL_Record :=
      (Name   => "Johnson, John       ",
       SSN    => "111223333",
       Salary => Salary_Conversions.To_Binary(Some_Salary));

begin                                                                           110
   Prog (Some_Record);
   ...
end Test_Call;

with Interfaces.COBOL;                                                          111
with COBOL_Sequential_IO; -- Assumed to be supplied by implementation
procedure Test_External_Formats is

{AI12-0442-1}     -- Using data created by a COBOL program                      112/5
   -- Assume that a COBOL program has created a sequential file with
   --  the following record structure, and that we want
   --  process the records in an Ada program
   --  01 EMPLOYEE-RECORD
   --    05 NAME    PIC X(20).
   --    05 SSN     PIC X(9).
   --    05 SALARY  PIC 99999V99 USAGE COMP.
   --    05 ADJUST  PIC S999V999 SIGN LEADING SEPARATE.
   -- The COMP data is binary (32 bits), high-order byte first
```

```
   package COBOL renames Interfaces.COBOL;

   type Salary_Type      is delta 0.01  digits 7;
   type Adjustments_Type is delta 0.001 digits 6;

{AI05-0229-1}    type COBOL_Employee_Record_Type is  -- External representation
      record
         Name    : COBOL.Alphanumeric(1..20);
         SSN     : COBOL.Alphanumeric(1..9);
         Salary  : COBOL.Byte_Array(1..4);
         Adjust  : COBOL.Numeric(1..7);  -- Sign and 6 digits
      end record
      with Convention => COBOL;

   package COBOL_Employee_IO is
      new COBOL_Sequential_IO(COBOL_Employee_Record_Type);
   use COBOL_Employee_IO;

   COBOL_File : File_Type;

   type Ada_Employee_Record_Type is  -- Internal representation
      record
         Name    : String(1..20);
         SSN     : String(1..9);
         Salary  : Salary_Type;
         Adjust  : Adjustments_Type;
      end record;

   COBOL_Record : COBOL_Employee_Record_Type;
   Ada_Record   : Ada_Employee_Record_Type;

   package Salary_Conversions is
      new COBOL.Decimal_Conversions(Salary_Type);
   use Salary_Conversions;

   package Adjustments_Conversions is
      new COBOL.Decimal_Conversions(Adjustments_Type);
   use Adjustments_Conversions;

begin
   Open (COBOL_File, Name => "Some_File");

   loop
     Read (COBOL_File, COBOL_Record);

{AI12-0178-1}      Ada_Record.Name := COBOL.To_Ada(COBOL_Record.Name);
     Ada_Record.SSN  := COBOL.To_Ada(COBOL_Record.SSN);
     Ada_Record.Salary :=
        To_Decimal(COBOL_Record.Salary, COBOL.High_Order_First);
     Ada_Record.Adjust :=
        To_Decimal(COBOL_Record.Adjust, COBOL.Leading_Separate);
     ... -- Process Ada_Record
   end loop;
exception
   when End_Error => ...
end Test_External_Formats;
```

Wording Changes from Ada 95

{*8652/0066*} {*AI95-00071-01*} **Corrigendum:** Corrected the definition of Valid to match COBOL.

{*8652/0067*} {*AI95-00072-01*} **Corrigendum:** Specified the bounds of the results of To_Display, To_Packed, and To_Binary.

B.5 Interfacing with Fortran

{*AI05-0229-1*} The facilities relevant to interfacing with the Fortran language are the package Interfaces.Fortran and support for specifying the Convention aspect with *convention*_identifier Fortran.

The package Interfaces.Fortran defines Ada types whose representations are identical to the default representations of the Fortran intrinsic types Integer, Real, Double Precision, Complex, Logical, and

Character in a supported Fortran implementation. These Ada types can therefore be used to pass objects between Ada and Fortran programs.

Static Semantics

The library package Interfaces.Fortran has the following declaration: 3

```
{AI12-0414-1} with Ada.Numerics.Generic_Complex_Types;  -- see G.1.1                4/5
pragma Elaborate_All(Ada.Numerics.Generic_Complex_Types);
package Interfaces.Fortran
   with Pure is

   type Fortran_Integer is range implementation-defined;                             5

   type Real             is digits implementation-defined;                           6
   type Double_Precision is digits implementation-defined;

   type Logical is new Boolean;                                                      7

   package Single_Precision_Complex_Types is                                         8
      new Ada.Numerics.Generic_Complex_Types (Real);

   type Complex is new Single_Precision_Complex_Types.Complex;                       9

   subtype Imaginary is Single_Precision_Complex_Types.Imaginary;                    10
   i : Imaginary renames Single_Precision_Complex_Types.i;
   j : Imaginary renames Single_Precision_Complex_Types.j;

{AI12-0058-1}    package Double_Precision_Complex_Types is                          10.1/5
      new Ada.Numerics.Generic_Complex_Types (Double_Precision);

{AI12-0058-1}    type Double_Complex is new Double_Precision_Complex_Types.Complex; 10.2/5

{AI12-0058-1}    subtype Double_Imaginary is                                        10.3/5
Double_Precision_Complex_Types.Imaginary;

   type Character_Set is implementation-defined character type;                      11

{AI05-0229-1}    type Fortran_Character is array (Positive range <>) of             12/3
Character_Set
      with Pack;

   function To_Fortran (Item : in Character) return Character_Set;                   13
   function To_Ada (Item : in Character_Set) return Character;

   function To_Fortran (Item : in String) return Fortran_Character;                  14
   function To_Ada     (Item : in Fortran_Character) return String;

   procedure To_Fortran (Item       : in String;                                     15
                         Target     : out Fortran_Character;
                         Last       : out Natural);

   procedure To_Ada (Item     : in Fortran_Character;                                16
                     Target   : out String;
                     Last     : out Natural);

end Interfaces.Fortran;                                                              17
```

Implementation defined: The types Fortran_Integer, Real, Double_Precision, and Character_Set in Interfaces.Fortran. 17.a.1/1

Ramification: {*AI12-0058-1*} The means by which the Complex and Double_Complex types are provided in Interfaces.Fortran creates a dependence of Interfaces.Fortran on Numerics.Generic_Complex_Types (see G.1.1). This dependence is intentional and unavoidable, if the Fortran-compatible Complex and Double_Complex types are to be useful in Ada code without duplicating facilities defined elsewhere. 17.a/5

{*AI12-0058-1*} The types Fortran_Integer, Real, Double_Precision, Logical, Complex, Double_Complex, Character_Set, and Fortran_Character are Fortran-compatible. 18/5

The To_Fortran and To_Ada functions map between the Ada type Character and the Fortran type Character_Set, and also between the Ada type String and the Fortran type Fortran_Character. The To_Fortran and To_Ada procedures have analogous effects to the string conversion subprograms found in Interfaces.COBOL. 19

Implementation Requirements

20/3 {*AI05-0229-1*} An implementation shall support specifying aspect Convention with a Fortran *convention*_identifier for a Fortran-eligible type (see B.1).

Implementation Permissions

21/5 {*AI12-0058-1*} {*AI12-0263-1*} {*AI12-0450-1*} An implementation may add additional declarations to the Fortran interface packages. For example, declarations are permitted for the character types corresponding to Fortran character kinds 'ascii' and 'iso_10646', which in turn correspond to ISO/IEC 646:1991 and to UCS-4 as specified in ISO/IEC 10646:2020.

21.a.1/5 **Reason:** {*AI12-0058-1*} Fortran compilers are required to recognize 'ascii' and 'iso_10646' as arguments to the SELECTED_CHAR_KIND intrinsic function, but are not required to support those kinds.

21.a/5 **Discussion:** {*AI12-0058-1*} Implementations may add auxiliary declarations as needed to assist in the declarations of additional Fortran-compatible types. For example, if a wide character type is defined to match a Fortran 90 wide character type (accessible in Fortran 90 with the Kind attribute), then an auxiliary character set may be declared to serve as its component type.

Implementation Advice

22 An Ada implementation should support the following interface correspondences between Ada and Fortran:

23 • An Ada procedure corresponds to a Fortran subroutine.

24 • An Ada function corresponds to a Fortran function.

25 • An Ada parameter of an elementary, array, or record type T is passed as a T_F argument to a Fortran procedure, where T_F is the Fortran type corresponding to the Ada type T, and where the INTENT attribute of the corresponding dummy argument matches the Ada formal parameter mode; the Fortran implementation's parameter passing conventions are used. For elementary types, a local copy is used if necessary to ensure by-copy semantics.

26 • An Ada parameter of an access-to-subprogram type is passed as a reference to a Fortran procedure whose interface corresponds to the designated subprogram's specification.

26.a/2 **Implementation Advice:** If Fortran interfacing is supported, the interface correspondences between Ada and Fortran should be supported.

27 NOTE 1 An object of a Fortran-compatible record type, declared in a library package or subprogram, can correspond to a Fortran common block; the type also corresponds to a Fortran “derived type”.

28/5 NOTE 2 {*AI12-0224-1*} For Fortran facilities not addressed by this subclause, consider using the Fortran to C interoperability features defined in ISO/IEC 1594-1:2018 along with the C interfacing features defined in B.3.

Examples

29 *Example of Interfaces.Fortran:*

30
```
with Interfaces.Fortran;
use Interfaces.Fortran;
procedure Ada_Application is
```

31/5
```
{AI05-0229-1} {AI12-0178-1}    type Fortran_Matrix is
      array (Fortran_Integer range <>,
             Fortran_Integer range <>) of Double_Precision
      with Convention => Fortran;                    -- stored in Fortran's
                                                     -- column-major order
   procedure Invert (Rank : in Fortran_Integer; X : in out Fortran_Matrix)
      with Import => True, Convention => Fortran; -- a Fortran subroutine
```

32
```
   Rank      : constant Fortran_Integer := 100;
   My_Matrix : Fortran_Matrix (1 .. Rank, 1 .. Rank);
```

```
{AI12-0058-1}    Precision: constant := 6;                                    33/5
   type Standard_Deviation is digits Precision
      with Convention => Fortran;
   Deviation : Standard_Deviation;
      -- Declarations to match the following Fortran declarations:
      --   integer, parameter :: precision = selected_real_kind(p=6)
      --   real(precision) :: deviation
begin                                                                         34
   ...                                                                        35
   My_Matrix := ...;
   ...
   Invert (Rank, My_Matrix);
   ...
{AI12-0058-1}    Deviation := ...;                                           36/5
   ...
end Ada_Application;                                                          37
```

Wording Changes from Ada 2012

{*AI12-0058-1*} **Correction:** The package Double_Precision_Complex_Types and associated types are added to package Interfaces.Fortran. In unusual circumstances, this could cause an incompatibility; we don't document it as an incompatibility as implementations are allowed to add declarations to this package, so that risk of an incompatibility is present for any move from one version of an implementation to another (not to mention to another implementation). As such, the language-defined additions make no change in the risk of incompatibility. 37.a/5

Annex C
(normative)
Systems Programming

[The Systems Programming Annex specifies additional capabilities provided for low-level programming. These capabilities are also required in many real-time, embedded, distributed, and information systems.] 1

Extensions to Ada 83

This Annex is new to Ada 95. 1.a

C.1 Access to Machine Operations

{*AI05-0299-1*} [This subclause specifies rules regarding access to machine instructions from within an Ada program.] 1/3

Implementation defined: Implementation-defined intrinsic subprograms. 1.a/2

Implementation Requirements

{*AI12-0320-1*} The implementation shall support machine code insertions (see 13.8) or intrinsic subprograms (see 6.3.1) (or both). The implementation shall allow the use of Ada entities as operands for such machine code insertions or intrinsic subprograms. 2/5

Discussion: {*AI12-0320-1*} Ada entities could be used as operands in machine code insertions and/or intrinsic subprograms using language-defined attributes (such as address), implementation-defined attributes, or could be directly allowed. 2.a/5

Implementation Advice

The machine code or intrinsics support should allow access to all operations normally available to assembly language programmers for the target environment, including privileged instructions, if any. 3

Implementation Advice: The machine code or intrinsics support should allow access to all operations normally available to assembly language programmers for the target environment. 3.a.1/2

Ramification: Of course, on a machine with protection, an attempt to execute a privileged instruction in user mode will probably trap. Nonetheless, we want implementations to provide access to them so that Ada can be used to write systems programs that run in privileged mode. 3.a

{*AI05-0229-1*} The support for interfacing aspects (see Annex B) should include interface to assembler; the default assembler should be associated with the convention identifier Assembler. 4/3

Implementation Advice: Interface to assembler should be supported; the default assembler should be associated with the convention identifier Assembler. 4.a/2

If an entity is exported to assembly language, then the implementation should allocate it at an addressable location, and should ensure that it is retained by the linking process, even if not otherwise referenced from the Ada code. The implementation should assume that any call to a machine code or assembler subprogram is allowed to read or update every object that is specified as exported. 5

Implementation Advice: If an entity is exported to assembly language, then the implementation should allocate it at an addressable location even if not otherwise referenced from the Ada code. A call to a machine code or assembler subprogram should be treated as if it can read or update every object that is specified as exported. 5.a/5

Documentation Requirements

The implementation shall document the overhead associated with calling machine-code or intrinsic subprograms, as compared to a fully-inlined call, and to a regular out-of-line call. 6

6.a/2 **Documentation Requirement:** The overhead of calling machine-code or intrinsic subprograms.

7 The implementation shall document the types of the package System.Machine_Code usable for machine code insertions, and the attributes to be used in machine code insertions for references to Ada entities.

7.a/2 **Documentation Requirement:** The types and attributes used in machine code insertions.

8/3 {*AI05-0229-1*} The implementation shall document the subprogram calling conventions associated with the convention identifiers available for use with the Convention aspect (Ada and Assembler, at a minimum), including register saving, exception propagation, parameter passing, and function value returning.

8.a/2 **Documentation Requirement:** The subprogram calling conventions for all supported convention identifiers.

9 For exported and imported subprograms, the implementation shall document the mapping between the Link_Name string, if specified, or the Ada designator, if not, and the external link name used for such a subprogram.

9.a/2 *This paragraph was deleted.*

9.b/2 **Documentation Requirement:** The mapping between the Link_Name or Ada designator and the external link name.

Implementation Advice

10 The implementation should ensure that little or no overhead is associated with calling intrinsic and machine-code subprograms.

10.a/2 **Implementation Advice:** Little or no overhead should be associated with calling intrinsic and machine-code subprograms.

11 It is recommended that intrinsic subprograms be provided for convenient access to any machine operations that provide special capabilities or efficiency and that are not otherwise available through the language constructs. Examples of such instructions include:

12/5 • {*AI12-0453-1*} Atomic read-modify-write operations — for example, test and set, compare and swap, decrement and test, enqueue/dequeue.

13/5 • {*AI12-0453-1*} Standard numeric functions — for example, *sin*, *log*.

14/5 • {*AI12-0453-1*} String manipulation operations — for example, translate and test.

15/5 • {*AI12-0453-1*} Vector operations — for example, compare vector against thresholds.

16 • Direct operations on I/O ports.

16.a/2 **Implementation Advice:** Intrinsic subprograms should be provided to access any machine operations that provide special capabilities or efficiency not normally available.

C.2 Required Representation Support

1/3 {*AI95-00434-01*} {*AI05-0299-1*} This subclause specifies minimal requirements on the support for representation items and related features.

Implementation Requirements

2 The implementation shall support at least the functionality defined by the recommended levels of support in Clause 13.

C.3 Interrupt Support

1/3 {*AI05-0299-1*} [This subclause specifies the language-defined model for hardware interrupts in addition to mechanisms for handling interrupts.]

Dynamic Semantics

[An *interrupt* represents a class of events that are detected by the hardware or the system software.] Interrupts are said to occur. An *occurrence* of an interrupt is separable into generation and delivery. *Generation* of an interrupt is the event in the underlying hardware or system that makes the interrupt available to the program. *Delivery* is the action that invokes part of the program as response to the interrupt occurrence. Between generation and delivery, the interrupt occurrence [(or interrupt)] is *pending*. Some or all interrupts may be *blocked*. When an interrupt is blocked, all occurrences of that interrupt are prevented from being delivered. Certain interrupts are *reserved*. The set of reserved interrupts is implementation defined. A reserved interrupt is either an interrupt for which user-defined handlers are not supported, or one which already has an attached handler by some other implementation-defined means. Program units can be connected to nonreserved interrupts. While connected, the program unit is said to be *attached* to that interrupt. The execution of that program unit, the *interrupt handler*, is invoked upon delivery of the interrupt occurrence. 2

This paragraph was deleted. 2.a/2

To be honest: As an obsolescent feature, interrupts may be attached to task entries by an address clause. See J.7.1. 2.b

While a handler is attached to an interrupt, it is called once for each delivered occurrence of that interrupt. While the handler executes, the corresponding interrupt is blocked. 3

While an interrupt is blocked, all occurrences of that interrupt are prevented from being delivered. Whether such occurrences remain pending or are lost is implementation defined. 4

Each interrupt has a *default treatment* which determines the system's response to an occurrence of that interrupt when no user-defined handler is attached. The set of possible default treatments is implementation defined, as is the method (if one exists) for configuring the default treatments for interrupts. 5

An interrupt is delivered to the handler (or default treatment) that is in effect for that interrupt at the time of delivery. 6

An exception propagated from a handler that is invoked by an interrupt has no effect. 7

[If the Ceiling_Locking policy (see D.3) is in effect, the interrupt handler executes with the active priority that is the ceiling priority of the corresponding protected object.] 8

Implementation Requirements

{*AI12-0445-1*} The implementation shall provide a mechanism to determine the minimum stack space that is necessary for each interrupt handler and to reserve that space for the execution of the handler. [This space should accommodate nested invocations of the handler where the system permits this.] 9/5

If the hardware or the underlying system holds pending interrupt occurrences, the implementation shall provide for later delivery of these occurrences to the program. 10

If the Ceiling_Locking policy is not in effect, the implementation shall provide means for the application to specify whether interrupts are to be blocked during protected actions. 11

Documentation Requirements

The implementation shall document the following items: 12

Discussion: This information may be different for different forms of interrupt handlers. 12.a

1. For each interrupt, which interrupts are blocked from delivery when a handler attached to that interrupt executes (either as a result of an interrupt delivery or of an ordinary call on a procedure of the corresponding protected object). 13

14 2. Any interrupts that cannot be blocked, and the effect of attaching handlers to such interrupts, if this is permitted.

15 3. Which run-time stack an interrupt handler uses when it executes as a result of an interrupt delivery; if this is configurable, what is the mechanism to do so; how to specify how much space to reserve on that stack.

16/5 4. {*AI12-0453-1*} Any implementation- or hardware-specific activity that happens before a user-defined interrupt handler gets control (for example, reading device registers, acknowledging devices).

17 5. Any timing or other limitations imposed on the execution of interrupt handlers.

18 6. The state (blocked/unblocked) of the nonreserved interrupts when the program starts; if some interrupts are unblocked, what is the mechanism a program can use to protect itself before it can attach the corresponding handlers.

19 7. Whether the interrupted task is allowed to resume execution before the interrupt handler returns.

20/5 8. {*AI12-0453-1*} The treatment of interrupt occurrences that are generated while the interrupt is blocked; that is, whether one or more occurrences are held for later delivery, or all are lost.

21 9. Whether predefined or implementation-defined exceptions are raised as a result of the occurrence of any interrupt, and the mapping between the machine interrupts (or traps) and the predefined exceptions.

22 10. On a multi-processor, the rules governing the delivery of an interrupt to a particular processor.

22.a/2 **Documentation Requirement:** The treatment of interrupts.

Implementation Permissions

23/2 {*AI95-00434-01*} If the underlying system or hardware does not allow interrupts to be blocked, then no blocking is required [as part of the execution of subprograms of a protected object for which one of its subprograms is an interrupt handler].

24 In a multi-processor with more than one interrupt subsystem, it is implementation defined whether (and how) interrupt sources from separate subsystems share the same Interrupt_Id type (see C.3.2). In particular, the meaning of a blocked or pending interrupt may then be applicable to one processor only.

24.a **Discussion:** This issue is tightly related to the issue of scheduling on a multi-processor. In a sense, if a particular interrupt source is not available to all processors, the system is not truly homogeneous.

24.b/5 {*AI12-0453-1*} One way to approach this problem is to assign sub-ranges within Interrupt_Id to each interrupt subsystem, such that “similar” interrupt sources (for example a timer) in different subsystems get a distinct id.

25 Implementations are allowed to impose timing or other limitations on the execution of interrupt handlers.

25.a **Reason:** These limitations are often necessary to ensure proper behavior of the implementation.

26/3 {*AI95-00434-01*} {*AI05-0299-1*} Other forms of handlers are allowed to be supported, in which case the rules of this subclause should be adhered to.

27 The active priority of the execution of an interrupt handler is allowed to vary from one occurrence of the same interrupt to another.

Implementation Advice

28/2 {*AI95-00434-01*} If the Ceiling_Locking policy is not in effect, the implementation should provide means for the application to specify which interrupts are to be blocked during protected actions, if the underlying system allows for finer-grained control of interrupt blocking.

Implementation Advice: If the Ceiling_Locking policy is not in effect and the target system allows for finer-grained control of interrupt blocking, a means for the application to specify which interrupts are to be blocked during protected actions should be provided. 28.a/2

NOTE 1 {*AI12-0447-1*} The default treatment for an interrupt can be to keep the interrupt pending or to deliver it to an implementation-defined handler. Examples of actions that an implementation-defined handler can perform include aborting the partition, ignoring (that is, discarding occurrences of) the interrupt, or queuing one or more occurrences of the interrupt for possible later delivery when a user-defined handler is attached to that interrupt. 29/5

NOTE 2 It is a bounded error to call Task_Identification.Current_Task (see C.7.1) from an interrupt handler. 30

NOTE 3 The rule that an exception propagated from an interrupt handler has no effect is modeled after the rule about exceptions propagated out of task bodies. 31

C.3.1 Protected Procedure Handlers

Paragraphs 1 through 6 were moved to Annex J, "Obsolescent Features".

Static Semantics

{*AI05-0229-1*} For a parameterless protected procedure, the following language-defined representation aspects may be specified: 6.1/3

Interrupt_Handler 6.2/3

The type of aspect Interrupt_Handler is Boolean. If directly specified, the aspect_definition shall be a static expression. [This aspect is never inherited;] if not directly specified, the aspect is False.

Aspect Description for Interrupt_Handler: Protected procedure may be attached to interrupts. 6.a/3

Attach_Handler 6.3/3

The aspect Attach_Handler is an expression, which shall be of type Interrupts.Interrupt_Id. [This aspect is never inherited.]

Aspect Description for Attach_Handler: Protected procedure is attached to an interrupt. 6.b/3

Legality Rules

{*AI95-00434-01*} {*AI05-0033-1*} {*AI05-0229-1*} If either the Attach_Handler or Interrupt_Handler aspect are specified for a protected procedure, the corresponding protected_type_declaration or single_-protected_declaration shall be a library-level declaration and shall not be declared within a generic body. In addition to the places where Legality Rules normally apply (see 12.3), this rule also applies in the private part of an instance of a generic unit. 7/3

Discussion: In the case of a protected_type_declaration, an object_declaration of an object of that type need not be at library level. 7.a

{*AI05-0033-1*} {*AI05-0229-1*} We cannot allow these aspects in protected declarations in a generic body, because legality rules are not checked for instance bodies, and these should not be allowed if the instance is not at the library level. The protected types can be declared in the private part if this is desired. Note that while the 'Access to use the handler would provide the check in the case of Interrupt_Handler, there is no other check for Attach_Handler. Since these aspects are so similar, we want the rules to be the same. 7.b/3

This paragraph was deleted.{*AI95-00253-01*} {*AI95-00303-01*} {*AI05-0033-1*} 8/3

Dynamic Semantics

{*AI05-0229-1*} If the Interrupt_Handler aspect of a protected procedure is True, then the procedure may be attached dynamically, as a handler, to interrupts (see C.3.2). [Such procedures are allowed to be attached to multiple interrupts.] 9/3

{*AI05-0229-1*} The expression specified for the Attach_Handler aspect of a protected procedure *P* is evaluated as part of the creation of the protected object that contains *P*. The value of the expression identifies an interrupt. As part of the initialization of that object, *P* (the *handler* procedure) is attached to 10/3

the identified interrupt. A check is made that the corresponding interrupt is not reserved. Program_Error is raised if the check fails, and the existing treatment for the interrupt is not affected.

11/3 {*AI95-00434-01*} {*AI05-0229-1*} If the Ceiling_Locking policy (see D.3) is in effect, then upon the initialization of a protected object that contains a protected procedure for which either the Attach_Handler aspect is specified or the Interrupt_Handler aspect is True, a check is made that the initial ceiling priority of the object is in the range of System.Interrupt_Priority. If the check fails, Program_Error is raised.

12/3 {*8652/0068*} {*AI95-00121-01*} {*AI05-0229-1*} When a protected object is finalized, for any of its procedures that are attached to interrupts, the handler is detached. If the handler was attached by a procedure in the Interrupts package or if no user handler was previously attached to the interrupt, the default treatment is restored. If the Attach_Handler aspect was specified and the most recently attached handler for the same interrupt is the same as the one that was attached at the time the protected object was initialized, the previous handler is restored.

12.a/3 **Discussion:** {*8652/0068*} {*AI95-00121-01*} {*AI95-00303-01*} {*AI05-0229-1*} If all protected objects for interrupt handlers are declared at the library level, the finalization discussed above occurs only as part of the finalization of all library-level packages in a partition. However, objects of a protected type containing procedures with an Attach_Handler aspect specified need not be at the library level. Thus, an implementation needs to be able to restore handlers during the execution of the program. (An object with an Interrupt_Handler aspect also need not be at the library level, but such a handler cannot be attached to an interrupt using the Interrupts package.)

13 When a handler is attached to an interrupt, the interrupt is blocked [(subject to the Implementation Permission in C.3)] during the execution of every protected action on the protected object containing the handler.

13.1/5 {*AI12-0252-1*} If restriction No_Dynamic_Attachment is in effect, then a check is made that the interrupt identified by an Attach_Handler aspect does not appear in any previously elaborated Attach_Handler aspect; Program_Error is raised if this check fails.

13.a/5 **Reason:** When No_Dynamic_Attachment is in effect, it is not possible to call any previous handler. Therefore, the only possibility is to replace the original handler, rendering it ineffective. Having an unused handler in a program is likely a bug, so we require an exception to be raised.

Erroneous Execution

14 If the Ceiling_Locking policy (see D.3) is in effect and an interrupt is delivered to a handler, and the interrupt hardware priority is higher than the ceiling priority of the corresponding protected object, the execution of the program is erroneous.

14.1/3 {*8652/0068*} {*AI95-00121-01*} {*AI05-0229-1*} If the handlers for a given interrupt attached via aspect Attach_Handler are not attached and detached in a stack-like (LIFO) order, program execution is erroneous. In particular, when a protected object is finalized, the execution is erroneous if any of the procedures of the protected object are attached to interrupts via aspect Attach_Handler and the most recently attached handler for the same interrupt is not the same as the one that was attached at the time the protected object was initialized.

14.a/3 **Discussion:** {*8652/0068*} {*AI95-00121-01*} {*AI05-0229-1*} This simplifies implementation of the Attach_Handler aspect by not requiring a check that the current handler is the same as the one attached by the initialization of a protected object.

Metrics

15 The following metric shall be documented by the implementation:

16/2 • {*AI95-00434-01*} The worst-case overhead for an interrupt handler that is a parameterless protected procedure, in clock cycles. This is the execution time not directly attributable to the handler procedure or the interrupted execution. It is estimated as C – (A+B), where A is how long it takes to complete a given sequence of instructions without any interrupt, B is how long it takes to complete a normal call to a given protected procedure, and C is how long it takes to complete

the same sequence of instructions when it is interrupted by one execution of the same procedure called via an interrupt.

Implementation Note: The instruction sequence and interrupt handler used to measure interrupt handling overhead should be chosen so as to maximize the execution time cost due to cache misses. For example, if the processor has cache memory and the activity of an interrupt handler could invalidate the contents of cache memory, the handler should be written such that it invalidates all of the cache memory. 16.a

Documentation Requirement: The metrics for interrupt handlers. 16.b/2

Implementation Permissions

{*AI05-0229-1*} When the aspects Attach_Handler or Interrupt_Handler are specified for a protected procedure, the implementation is allowed to impose implementation-defined restrictions on the corresponding protected_type_declaration and protected_body. 17/3

Ramification: The restrictions may be on the constructs that are allowed within them, and on ordinary calls (i.e. not via interrupts) on protected operations in these protected objects. 17.a

Implementation defined: Any restrictions on a protected procedure or its containing type when an aspect Attach_handler or Interrupt_Handler is specified. 17.b/3

An implementation may use a different mechanism for invoking a protected procedure in response to a hardware interrupt than is used for a call to that protected procedure from a task. 18

Discussion: This is despite the fact that the priority of an interrupt handler (see D.1) is modeled after a hardware task calling the handler. 18.a

{*AI05-0229-1*} Notwithstanding what this subclause says elsewhere, the Attach_Handler and Interrupt_Handler aspects are allowed to be used for other, implementation defined, forms of interrupt handlers. 19/3

Ramification: {*AI05-0229-1*} For example, if an implementation wishes to allow interrupt handlers to have parameters, it is allowed to do so via these aspects; it need not invent implementation-defined aspects for the purpose. 19.a/3

Implementation defined: Any other forms of interrupt handler supported by the Attach_Handler and Interrupt_Handler aspects. 19.b/3

Implementation Advice

Whenever possible, the implementation should allow interrupt handlers to be called directly by the hardware. 20

Implementation Advice: Interrupt handlers should be called directly by the hardware. 20.a/2

Whenever practical, the implementation should detect violations of any implementation-defined restrictions before run time. 21

Implementation Advice: Violations of any implementation-defined restrictions on interrupt handlers should be detected before run time. 21.a/2

NOTE 1 {*AI05-0229-1*} {*AI12-0440-1*} The Attach_Handler aspect can provide static attachment of handlers to interrupts if the implementation supports preelaboration of protected objects. (See C.4.) 22/5

NOTE 2 {*AI95-00434-01*} {*AI12-0442-1*} For a protected object that has a (protected) procedure attached to an interrupt, the correct ceiling priority is at least as high as the highest processor priority at which that interrupt will ever be delivered. 23/5

NOTE 3 Protected procedures can also be attached dynamically to interrupts via operations declared in the predefined package Interrupts. 24

NOTE 4 An example of a possible implementation-defined restriction is disallowing the use of the standard storage pools within the body of a protected procedure that is an interrupt handler. 25

Incompatibilities With Ada 95

{*AI95-00253-01*} **Amendment Correction:** Corrected the wording so that the rules for the use of Attach_Handler and Interrupt_Handler are identical. This means that uses of pragma Interrupt_Handler outside of the target protected type or single protected object are now illegal. 25.a/2

Wording Changes from Ada 95

25.b/2 {*8652/0068*} {*AI95-00121-01*} **Corrigendum:** Clarified the meaning of "the previous handler" when finalizing protected objects containing interrupt handlers.

25.c/2 {*AI95-00303-01*} Dropped the requirement that an object of a type containing an Interrupt_Handler pragma must be declared at the library level. This was a generic contract model violation. This change is not an extension, as an attempt to attach such a handler with a routine in package Interrupts will fail an accessibility check anyway. Moreover, implementations can retain the rule as an implementation-defined restriction on the use of the type, as permitted by the Implementation Permissions above.

Extensions to Ada 2005

25.d/3 {*AI05-0229-1*} Aspects Interrupt_Handler and Attach_Handler are new; pragmas Interrupt_Handler and Attach_Handler are now obsolescent.

Wording Changes from Ada 2005

25.e/3 {*AI05-0033-1*} **Correction:** Added missing generic contract wording for the aspects Attach_Handler and Interrupt_Handler.

Inconsistencies With Ada 2012

25.f/5 {*AI12-0252-1*} **Correction:** Program_Error might be raised if two handlers are attached when restriction No_Dynamic_Attachment is in effect (including as part of the Ravenscar profile). Original Ada 2012 would have just ignored the first handler. This is almost certainly a bug rather than an intended result.

C.3.2 The Package Interrupts

Static Semantics

1 The following language-defined packages exist:

2/5
```
{AI05-0167-1} {AI12-0241-1} {AI12-0302-1} with System;
with System.Multiprocessors;
package Ada.Interrupts
   with Nonblocking, Global => in out synchronized is
   type Interrupt_Id is implementation-defined;
   type Parameterless_Handler is
      access protected procedure
      with Nonblocking => False;
```

3/1 *This paragraph was deleted.*

4
```
   function Is_Reserved (Interrupt : Interrupt_Id)
      return Boolean;
```

5
```
   function Is_Attached (Interrupt : Interrupt_Id)
      return Boolean;
```

6
```
   function Current_Handler (Interrupt : Interrupt_Id)
      return Parameterless_Handler;
```

7
```
   procedure Attach_Handler
      (New_Handler : in Parameterless_Handler;
       Interrupt   : in Interrupt_Id);
```

8
```
   procedure Exchange_Handler
      (Old_Handler : out Parameterless_Handler;
       New_Handler : in Parameterless_Handler;
       Interrupt   : in Interrupt_Id);
```

9
```
   procedure Detach_Handler
      (Interrupt : in Interrupt_Id);
```

10
```
   function Reference (Interrupt : Interrupt_Id)
      return System.Address;
```

10.1/3
```
   {AI05-0167-1}    function Get_CPU (Interrupt : Interrupt_Id)
      return System.Multiprocessors.CPU_Range;
```

```
private                                                                    11
   ... -- not specified by the language
end Ada.Interrupts;
```

```
{AI12-0302-1} package Ada.Interrupts.Names                                 12/5
   with Nonblocking, Global => null is
   implementation-defined : constant Interrupt_Id :=
      implementation-defined;
         . . .
   implementation-defined : constant Interrupt_Id :=
      implementation-defined;
end Ada.Interrupts.Names;
```

Dynamic Semantics

The Interrupt_Id type is an implementation-defined discrete type used to identify interrupts. 13

The Is_Reserved function returns True if and only if the specified interrupt is reserved. 14

The Is_Attached function returns True if and only if a user-specified interrupt handler is attached to the interrupt. 15

{*8652/0069*} {*AI95-00166-01*} The Current_Handler function returns a value that represents the attached handler of the interrupt. If no user-defined handler is attached to the interrupt, Current_Handler returns **null**. 16/1

{*AI05-0229-1*} The Attach_Handler procedure attaches the specified handler to the interrupt, overriding any existing treatment (including a user handler) in effect for that interrupt. If New_Handler is **null**, the default treatment is restored. If New_Handler designates a protected procedure for which the aspect Interrupt_Handler is False, Program_Error is raised. In this case, the operation does not modify the existing interrupt treatment. 17/3

{*8652/0069*} {*AI95-00166-01*} The Exchange_Handler procedure operates in the same manner as Attach_Handler with the addition that the value returned in Old_Handler designates the previous treatment for the specified interrupt. If the previous treatment is not a user-defined handler, **null** is returned. 18/1

Ramification: Calling Attach_Handler or Exchange_Handler with this value for New_Handler restores the previous handler. 18.a

{*8652/0069*} {*AI95-00166-01*} If the application uses only parameterless procedures as handlers (other types of handlers may be provided by the implementation, but are not required by the standard), then if Old_Handler is not **null**, it may be called to execute the previous handler. This provides a way to cascade application interrupt handlers. However, the default handler cannot be cascaded this way (Old_Handler must be **null** for the default handler). 18.a.1/1

The Detach_Handler procedure restores the default treatment for the specified interrupt. 19

For all operations defined in this package that take a parameter of type Interrupt_Id, with the exception of Is_Reserved and Reference, a check is made that the specified interrupt is not reserved. Program_Error is raised if this check fails. 20

{*AI05-0229-1*} If, by using the Attach_Handler, Detach_Handler, or Exchange_Handler procedures, an attempt is made to detach a handler that was attached statically (using the aspect Attach_Handler), the handler is not detached and Program_Error is raised. 21/3

{*AI95-00434-01*} The Reference function returns a value of type System.Address that can be used to attach a task entry via an address clause (see J.7.1) to the interrupt specified by Interrupt. This function raises Program_Error if attaching task entries to interrupts (or to this particular interrupt) is not supported. 22/2

{*AI05-0153-3*} The function Get_CPU returns the processor on which the handler for Interrupt is executed. If the handler can execute on more than one processor the value System.Multiprocessors.Not_A_Specific_CPU is returned. 22.1/3

Implementation Requirements

23 At no time during attachment or exchange of handlers shall the current handler of the corresponding interrupt be undefined.

Documentation Requirements

24/5 {*AI95-00434-01*} {*AI05-0229-1*} {*AI12-0320-1*} {*AI12-0444-1*} The implementation shall document, when the Ceiling_Locking policy (see D.3) is in effect, the default ceiling priority assigned to a protected object that contains a protected procedure that specifies either the Attach_Handler or Interrupt_Handler aspects, but does not specify the Interrupt_Priority aspect. [This default can be different for different interrupts.]

24.a.1/3 **Documentation Requirement:** If the Ceiling_Locking policy is in effect, the default ceiling priority for a protected object that specifies an interrupt handler aspect.

Implementation Advice

25 If implementation-defined forms of interrupt handler procedures are supported, such as protected procedures with parameters, then for each such form of a handler, a type analogous to Parameterless_Handler should be specified in a child package of Interrupts, with the same operations as in the predefined package Interrupts.

25.a/2 **Implementation Advice:** If implementation-defined forms of interrupt handler procedures are supported, then for each such form of a handler, a type analogous to Parameterless_Handler should be specified in a child package of Interrupts, with the same operations as in the predefined package Interrupts.

26 NOTE The package Interrupts.Names contains implementation-defined names (and constant values) for the interrupts that are supported by the implementation.

Examples

27 *Example of interrupt handlers:*

28/5
```
{AI05-0229-1} {AI12-0178-1} Device_Priority : constant
  array (Ada.Interrupts.Interrupt_Id range 1..5) of
    System.Interrupt_Priority := ( ... );
protected type Device_Interface
  (Int_Id : Ada.Interrupts.Interrupt_Id)
     with Interrupt_Priority => Device_Priority(Int_Id) is
  procedure Handler
     with Attach_Handler => Int_Id;
  ...
end Device_Interface;
  ...
Device_1_Driver : Device_Interface(1);
  ...
Device_5_Driver : Device_Interface(5);
  ...
```

Wording Changes from Ada 95

28.a/2 {*8652/0069*} {*AI95-00166-01*} **Corrigendum:** Clarified that the value returned by Current_Handler and Exchange_Handler for the default treatment is null.

Incompatibilities With Ada 2005

28.b/3 {*AI05-0167-1*} Function Get_CPU is added to Interrupts. If Interrupts is referenced in a use_clause, and an entity *E* with a defining_identifier of Get_CPU is defined in a package that is also referenced in a use_clause, the entity *E* may no longer be use-visible, resulting in errors. This should be rare and is easily fixed if it does occur.

C.4 Preelaboration Requirements

{*AI05-0299-1*} {*AI12-0417-1*} [This subclause specifies additional implementation and documentation requirements for the Preelaborate aspect (see 10.2.1).] 1/5

Implementation Requirements

The implementation shall not incur any run-time overhead for the elaboration checks of subprograms and protected_bodies declared in preelaborated library units. 2

The implementation shall not execute any memory write operations after load time for the elaboration of constant objects declared immediately within the declarative region of a preelaborated library package, so long as the subtype and initial expression (or default initial expressions if initialized by default) of the object_declaration satisfy the following restrictions. The meaning of *load time* is implementation defined. 3

Discussion: On systems where the image of the partition is initially copied from disk to RAM, or from ROM to RAM, prior to starting execution of the partition, the intention is that "load time" consist of this initial copying step. On other systems, load time and run time might actually be interspersed. 3.a

- Any subtype_mark denotes a statically constrained subtype, with statically constrained subcomponents, if any; 4
- {*AI95-00161-01*} no subtype_mark denotes a controlled type, a private type, a private extension, a generic formal private type, a generic formal derived type, or a descendant of such a type; 4.1/2

  Reason: For an implementation that uses the registration method of finalization, a controlled object will require some code executed to register the object at the appropriate point. The other types are those that *might* have a controlled component. None of these types were allowed in preelaborated units in Ada 95. These types are covered by the Implementation Advice, of course, so they should still execute as little code as possible. 4.a.1/2
- any constraint is a static constraint; 5
- any allocator is for an access-to-constant type; 6
- any uses of predefined operators appear only within static expressions; 7
- any primaries that are names, other than attribute_references for the Access or Address attributes, appear only within static expressions; 8

  Ramification: This cuts out attribute_references that are not static, except for Access and Address. 8.a
- {*AI12-0368-1*} any name that is not part of a static expression is an expanded name or direct_name that statically names some entity; 9/5

  Ramification: {*AI12-0368-1*} This cuts out function_calls and type_conversions that are not static, including calls on attribute functions like 'Image and 'Value. We do allow components if those components don't require any evaluation or checks. 9.a/5
- any discrete_choice of an array_aggregate is static; 10
- no language-defined check associated with the elaboration of the object_declaration can fail. 11

  Reason: {*AI95-00114-01*} The intent is that aggregates all of whose scalar subcomponents are static and all of whose access subcomponents are **null**, allocators for access-to-constant types, or X'Access, will be supported with no run-time code generated. 11.a/2

Documentation Requirements

The implementation shall document any circumstances under which the elaboration of a preelaborated package causes code to be executed at run time. 12

Documentation Requirement: Any circumstances when the elaboration of a preelaborated package causes code to be executed. 12.a/2

13 The implementation shall document whether the method used for initialization of preelaborated variables allows a partition to be restarted without reloading.

13.a.1/2 **Documentation Requirement:** Whether a partition can be restarted without reloading.

13.a/2 *This paragraph was deleted.*

13.b/2 **Discussion:** {*AI95-00114-01*} This covers the issue of the run-time system itself being restartable, so that need not be a separate Documentation Requirement.

Implementation Advice

14 It is recommended that preelaborated packages be implemented in such a way that there should be little or no code executed at run time for the elaboration of entities not already covered by the Implementation Requirements.

14.a/2 **Implementation Advice:** Preelaborated packages should be implemented such that little or no code is executed at run time for the elaboration of entities.

Wording Changes from Ada 95

14.b/2 {*AI95-00161-01*} Added wording to exclude the additional kinds of types allowed in preelaborated units from the Implementation Requirements.

Wording Changes from Ada 2012

14.c/5 {*AI12-0368-1*} Added wording to allow components so long as no evaluation or checks are required for the reference.

C.5 Aspect Discard_Names

1/4 {*AI12-0072-1*} [Specifying the aspect Discard_Names can be used to request a reduction in storage used for the names of entities with runtime name text.]

Static Semantics

1.1/4 {*AI12-0072-1*} An entity with *runtime name text* is a nonderived enumeration first subtype, a tagged first subtype, or an exception.

1.2/4 {*AI12-0072-1*} For an entity with runtime name text, the following language-defined representation aspect may be specified:

1.3/4 Discard_Names

The type of aspect Discard_Names is Boolean. If directly specified, the aspect_definition shall be a static expression. If not specified (including by inheritance), the aspect is False.

1.a.1/4 **Aspect Description for Discard_Names:** Requests a reduction in storage for names associated with an entity.

Syntax

2 The form of a pragma Discard_Names is as follows:

3 **pragma** Discard_Names[([On =>] local_name)];

4 A pragma Discard_Names is allowed only immediately within a declarative_part, immediately within a package_specification, or as a configuration pragma.

Legality Rules

5/4 {*AI12-0072-1*} The local_name (if present) shall denote an entity with runtime name text. The pragma specifies that the aspect Discard_Names for the type or exception has the value True. Without a local_name, the pragma specifies that all entities with runtime name text declared after the pragma, within the same declarative region have the value True for aspect Discard_Names. Alternatively, the pragma can be used as a configuration pragma. If the configuration pragma Discard_Names applies to a

compilation unit, all entities with runtime name text declared in the compilation unit have the value True for the aspect Discard_Names.

Ramification: {*AI12-0072-1*} If the aspect is specified for a type, then it is inherited by all descendants of the type. The aspect cannot be specified as False on a derived type (because specifying the aspect is not allowed on derived enumeration types, and by rule applying to all aspects for other types (see 13.1.1)). 5.a/4

Static Semantics

If a local_name is given, then a pragma Discard_Names is a representation pragma. 6

This paragraph was deleted. {*AI05-0229-1*} {*AI12-0072-1*} 6.a/4

{*AI95-00285-01*} {*AI95-00400-01*} {*AI12-0072-1*} If the aspect Discard_Names is True for an enumeration type, then the semantics of the Wide_Wide_Image and Wide_Wide_Value attributes are implementation defined for that type[; the semantics of Image, Wide_Image, Value, and Wide_Value are still defined in terms of Wide_Wide_Image and Wide_Wide_Value]. In addition, the semantics of Text_IO.Enumeration_IO are implementation defined. If the aspect Discard_Names is True for a tagged type, then the semantics of the Tags.Wide_Wide_Expanded_Name function are implementation defined for that type[; the semantics of Tags.Expanded_Name and Tags.Wide_Expanded_Name are still defined in terms of Tags.Wide_Wide_Expanded_Name]. If the aspect Discard_Names is True for an exception, then the semantics of the Exceptions.Wide_Wide_Exception_Name function are implementation defined for that exception[; the semantics of Exceptions.Exception_Name and Exceptions.Wide_Exception_Name are still defined in terms of Exceptions.Wide_Wide_Exception_Name]. 7/4

Implementation defined: The semantics of some attributes and functions of an entity for which aspect Discard_Names is True. 7.a/4

Ramification: The Width attribute is still defined in terms of Image. 7.b

{*AI95-00285-01*} {*AI12-0072-1*} The semantics of S'Wide_Wide_Image and S'Wide_Wide_Value are implementation defined for any subtype of an enumeration type for which the aspect is True. (The pragma, if used, actually names the first subtype, of course.) 7.c/4

Implementation Advice

{*AI12-0072-1*} If the aspect Discard_Names is True for an entity, then the implementation should reduce the amount of storage used for storing names associated with that entity. 8/4

Implementation Advice: If aspect Discard_Names is True for an entity, then the amount of storage used for storing names associated with that entity should be reduced. 8.a/4

Reason: A typical implementation of the Image attribute for enumeration types is to store a table containing the names of all the enumeration literals. Aspect Discard_Names allows the implementation to avoid storing such a table without having to prove that the Image attribute is never used (which can be difficult in the presence of separate compilation). 8.b

We did not specify the semantics of the Image attribute when aspect Discard_Names is True because different semantics might be desirable in different situations. In some cases, it might make sense to use the Image attribute to print out a useful value that can be used to identify the entity given information in compiler-generated listings. In other cases, it might make sense to get an error at compile time or at run time. In cases where memory is plentiful, the simplest implementation makes sense: ignore the aspect. Implementations that are capable of avoiding the extra storage in cases where the Image attribute is never used might also wish to ignore the aspect. 8.c

The same applies to the Tags.Expanded_Name and Exceptions.Exception_Name functions. 8.d

Wording Changes from Ada 95

{*AI95-00285-01*} {*AI95-00400-01*} Updated the wording to reflect that the double wide image and value functions are now the master versions that the others are defined from. 8.e/2

Wording Changes from Ada 2012

{*AI12-0072-1*} **Corrigendum:** Defined the pragma in terms of the aspect Discard_Names, and added a missing definition of the meaning of the configuration pragma. This is not intended to make any semantic change (Ada 2012 has an aspect Discard_Names defined via blanket rules for representation pragmas in 13.1 and 13.1.1), just to clarify the meaning. 8.f/4

C.6 Shared Variable Control

1/3 {*AI05-0229-1*} {*AI05-0299-1*} [This subclause defines representation aspects that control the use of shared variables.]

Paragraphs 2 through 6 were moved to Annex J, "Obsolescent Features".

Static Semantics

6.1/5 {*AI05-0229-1*} {*AI12-0282-1*} For an object_declaration, a component_declaration, a full_type_declaration, or a formal_complete_type_declaration, the following representation aspects may be specified:

6.2/3 Atomic The type of aspect Atomic is Boolean.

6.a/3 **Aspect Description for Atomic:** Declare that a type, object, or component is atomic.

6.3/3 Independent The type of aspect Independent is Boolean.

6.b/3 **Aspect Description for Independent:** Declare that a type, object, or component is independently addressable.

6.4/3 Volatile The type of aspect Volatile is Boolean.

6.c/3 **Aspect Description for Volatile:** Declare that a type, object, or component is volatile.

6.5/5 {*AI12-0363-1*} Full_Access_Only
The type of aspect Full_Access_Only is Boolean.

6.c.1/5 **Aspect Description for Full_Access_Only:** Declare that a volatile type, object, or component is full access.

6.6/5 {*AI05-0229-1*} {*AI12-0282-1*} For a full_type_declaration of an array type, an object_declaration for an object of an anonymous array type, or the formal_complete_type_declaration of a formal array type, the following representation aspects may be specified:

6.7/3 Atomic_Components
The type of aspect Atomic_Components is Boolean.

6.d/3 **Aspect Description for Atomic_Components:** Declare that the components of an array type or object are atomic.

6.8/3 Volatile_Components
The type of aspect Volatile_Components is Boolean.

6.e/3 **Aspect Description for Volatile_Components:** Declare that the components of an array type or object are volatile.

6.9/5 {*AI05-0229-1*} {*AI12-0282-1*} For a full_type_declaration of a composite type, an object_declaration for an object of an anonymous composite type, or the formal_complete_type_declaration of a formal composite type, the following representation aspect may be specified:

6.10/3 Independent_Components
The type of aspect Independent_Components is Boolean.

6.f/3 **Aspect Description for Independent_Components:** Declare that the components of an array or record type, or an array object, are independently addressable.

6.11/5 {*AI05-0229-1*} {*AI12-0363-1*} If any of these aspects are directly specified, the aspect_definition shall be a static expression. If not specified for a type (including by inheritance), the Atomic, Atomic_Components, and Full_Access_Only aspects are False. If any of these aspects are specified True for a type, then the corresponding aspect is True for all objects of the type. If the Atomic aspect is specified True, then the aspects Volatile, Independent, and Volatile_Component (if defined) are True; if the Atomic_Components aspect is specified True, then the aspects Volatile, Volatile_Components, and Independent_Components are True. If the Volatile aspect is specified True, then the Volatile_Components aspect (if defined) is True, and vice versa. When not determined by one of the

other aspects, or for an object by its type, the Volatile, Volatile_Components, Independent, and Independent_Components aspects are False.

Ramification: {*AI12-0363-1*} Aspects Volatile and Volatile_Components (when defined) are equivalent. We provide the Volatile_Components aspect only to give symmetry with Atomic_Components and Independent_Components aspects. 6.g/5

{*AI95-00272-01*} {*AI05-0229-1*} An *atomic* type is one for which the aspect Atomic is True. An *atomic* object (including a component) is one for which the aspect Atomic is True, or a component of an array for which the aspect Atomic_Components is True for the associated type, or any object of an atomic type, other than objects obtained by evaluating a slice. 7/3

Ramification: {*AI95-00272-01*} A slice of an atomic array object is not itself atomic. That's necessary as executing a read or write of a dynamic number of components in a single instruction is not possible on many targets. 7.a/2

{*AI05-0229-1*} A *volatile* type is one for which the aspect Volatile is True. A *volatile* object (including a component) is one for which the aspect Volatile is True, or a component of an array for which the aspect Volatile_Components is True for the associated type, or any object of a volatile type. In addition, every atomic type or object is also defined to be volatile. Finally, if an object is volatile, then so are all of its subcomponents [(the same does not apply to atomic)]. 8/3

{*AI05-0009-1*} {*AI05-0229-1*} {*AI12-0001-1*} When True, the aspects Independent and Independent_Components *specify as independently addressable* the named object or component(s), or in the case of a type, all objects or components of that type. All atomic objects and aliased objects are considered to be specified as independently addressable. 8.1/4

Ramification: If the compiler cannot guarantee that an object (including a component) for which aspect Independent or aspect Independent_Components is True is independently addressable from any other nonoverlapping object, then the aspect specification must be rejected. 8.a/3

Similarly, an atomic object (including atomic components) is always independently addressable from any other nonoverlapping object. Any representation item which would prevent this from being true should be rejected, notwithstanding what this Standard says elsewhere (specifically, in the Recommended Level of Support). 8.b/3

{*AI12-0363-1*} The Full_Access_Only aspect shall not be specified unless the associated type or object is volatile [(or atomic)]. A *full access* type is any atomic type, or a volatile type for which the aspect Full_Access_Only is True. A *full access* object (including a component) is any atomic object, or a volatile object for which the aspect Full_Access_Only is True for the object[or its type]. A Full_Access_Only aspect is illegal if any subcomponent of the object or type is a full access object or is of a generic formal type. 8.2/5

Ramification: {*AI12-0363-1*} This last rule breaks privacy, but that is considered OK for representation clauses when there is no clear alternative. Note that atomic objects may be nested, so long as the outer atomic object does not have the Full_Access_Only aspect True. 8.c/5

Reason: {*AI12-0363-1*} We disallow subcomponents of a generic formal type in a Full_Access_Only object or type as the actual to a formal type can be a full access type. We could have had a less restrictive rule, but such a use is unlikely as full access only objects are intended to be used to access memory-mapped devices with access restrictions, and those will need a concrete mapping not possible for generic formal types. 8.d/5

Paragraph 9 was moved to Annex J, "Obsolescent Features".

Legality Rules

{*AI05-0229-1*} If aspect Independent_Components is specified for a full_type_declaration, the declaration shall be that of an array or record type. 9.1/3

{*AI05-0229-1*} {*AI12-0001-1*} It is illegal to specify either of the aspects Atomic or Atomic_Components to have the value True for an object or type if the implementation cannot support the indivisible and independent reads and updates required by the aspect (see below). 10/4

11/4 {*AI12-0001-1*} It is illegal to specify the Size attribute of an atomic object, the Component_Size attribute for an array type with atomic components, or the layout attributes of an atomic component, in a way that prevents the implementation from performing the required indivisible and independent reads and updates.

12/5 {*AI05-0142-4*} {*AI05-0218-1*} {*AI12-0282-1*} {*AI12-0363-1*} If an atomic object is passed as a parameter, then the formal parameter shall either have an atomic type or allow pass by copy. If an atomic object is used as an actual for a generic formal object of mode **in out**, then the type of the generic formal object shall be atomic. If the prefix of an attribute_reference for an Access attribute denotes an atomic object [(including a component)], then the designated type of the resulting access type shall be atomic. Corresponding rules apply to volatile objects and to full access objects.

12.a/3 **Ramification:** {*AI05-0142-4*} A formal parameter allows pass by copy if it is not **aliased** and it is of a type that allows pass by copy (that is, is not a by-reference type).

12.1/5 {*AI12-0128-1*} {*AI12-0363-1*} If a nonatomic subcomponent of a full access object is passed as an actual parameter in a call then the formal parameter shall allow pass by copy (and, at run time, the parameter shall be passed by copy). A nonatomic subcomponent of a full access object shall not be used as an actual for a generic formal of mode **in out**. The prefix of an attribute_reference for an Access attribute shall not denote a nonatomic subcomponent of a full access object.

12.2/5 {*AI05-0218-1*} {*AI12-0282-1*} {*AI12-0363-1*} If the Atomic, Atomic_Components, Volatile, Volatile_Components, Independent, Independent_Components, or Full_Access_Only aspect is True for a generic formal type, then that aspect shall be True for the actual type. If an atomic type is used as an actual for a generic formal derived type, then the ancestor of the formal type shall be atomic. A corresponding rule applies to volatile types and similarly to full access types.

12.3/5 {*AI12-0363-1*} If a type with volatile components is used as an actual for a generic formal array type, then the components of the formal type shall be volatile. Furthermore, if the actual type has atomic components and the formal array type has aliased components, then the components of the formal array type shall also be atomic. A corresponding rule applies when the actual type has volatile full access components.

12.b/5 **Reason:** {*AI12-0363-1*} The limitations on formal array types are separate for volatile and atomic because of the fact that only volatility is carried down to all subcomponents of a volatile object, while atomicity is not. The goal of both limitations is that we don't want 'Access for an access type to produce a value that designates an object whose atomicity and volatility don't agree with that of the designated type of the access type. The above rules ensure that the generic “sees” the relevant volatility and atomicity.

13/3 {*AI05-0229-1*} If an aspect Volatile, Volatile_Components, Atomic, or Atomic_Components is directly specified to have the value True for a stand-alone constant object, then the aspect Import shall also be specified as True for it.

13.a **Ramification:** Hence, no initialization expression is allowed for such a constant. Note that a constant that is atomic or volatile because of its type is allowed.

13.b **Reason:** Stand-alone constants that are explicitly specified as Atomic or Volatile only make sense if they are being manipulated outside the Ada program. From the Ada perspective the object is read-only. Nevertheless, if imported and atomic or volatile, the implementation should presume it might be altered externally. For an imported stand-alone constant that is not atomic or volatile, the implementation can assume that it will not be altered.

13.c/3 **To be honest:** {*AI05-0218-1*} Volatile_Components and Atomic_Components actually are aspects of the anonymous array type; this rule only applies when the aspect is specified directly on the constant object and not when the (named) array type has the aspect.

13.1/3 {*AI05-0009-1*} {*AI05-0229-1*} It is illegal to specify the aspect Independent or Independent_Components as True for a component, object or type if the implementation cannot provide the independent addressability required by the aspect (see 9.10).

{*AI05-0009-1*} {*AI05-0229-1*} It is illegal to specify a representation aspect for a component, object or type for which the aspect Independent or Independent_Components is True, in a way that prevents the implementation from providing the independent addressability required by the aspect. 13.2/3

Paragraph 14 was moved to Annex J, "Obsolescent Features".

Dynamic Semantics

For an atomic object (including an atomic component) all reads and updates of the object as a whole are indivisible. 15

{*AI05-0117-1*} {*AI05-0275-1*} All tasks of the program (on all processors) that read or update volatile variables see the same order of updates to the variables. A use of an atomic variable or other mechanism may be necessary to avoid erroneous execution and to ensure that access to nonatomic volatile variables is sequential (see 9.10). 16/3

Implementation Note: {*AI05-0117-1*} {*AI05-0275-1*} To ensure this, on a multiprocessor, any read or update of an atomic object may require the use of an appropriate memory barrier. 16.a/3

Discussion: {*AI05-0275-1*} From 9.10 it follows that (in non-erroneous programs) accesses to variables, including those shared by multiple tasks, are always sequential. This guarantees that no task will ever see partial updates of any variable. For volatile variables (including atomic variables), the above rule additionally specifies that all tasks see the same order of updates. 16.b/3

{*AI05-0275-1*} If for a shared variable *X*, a read of *X* occurs sequentially after an update of *X*, then the read will return the updated value if *X* is volatile or atomic, but may or or may not return the updated value if *X* is nonvolatile. For nonvolatile accesses, a signaling action is needed in order to share the updated value. 16.c/3

{*AI05-0275-1*} Because accesses to the same atomic variable by different tasks establish a sequential order between the actions of those tasks, implementations may be required to emit memory barriers around such updates or use atomic instructions that imply such barriers. 16.d/3

Two actions are sequential (see 9.10) if each is the read or update of the same atomic object. 17

If a type is atomic or volatile and it is not a by-copy type, then the type is defined to be a by-reference type. If any subcomponent of a type is atomic or volatile, then the type is defined to be a by-reference type. 18

If an actual parameter is atomic or volatile, and the corresponding formal parameter is not, then the parameter is passed by copy. 19

Implementation Note: Note that in the case where such a parameter is normally passed by reference, a copy of the actual will have to be produced at the call-site, and a pointer to the copy passed to the formal parameter. If the actual is atomic, any copying has to use indivisible read on the way in, and indivisible write on the way out. 19.a

Reason: It has to be known at compile time whether an atomic or a volatile parameter is to be passed by copy or by reference. For some types, it is unspecified whether parameters are passed by copy or by reference. The above rules further specify the parameter passing rules involving atomic and volatile types and objects. 19.b

{*AI12-0128-1*} {*AI12-0347-1*} All reads of or writes to any nonatomic subcomponent of a full access object are performed by reading and/or writing all of the nearest enclosing full access object. 19.1/5

Implementation Note: {*AI12-0005-1*} {*AI12-0128-1*} } For example, if a 32-bit record object has four nonatomic components, each occupying one byte, then an assignment to one of those components might normally be implemented on some target machines via some sort of store_byte instruction; if the record object is an atomic full access object then instead a 32-bit read-modify-write must be performed. That read-modify-write need not be atomic, although the read and the write must each separately be atomic. Note that it doesn't matter whether the store_byte instruction would have executed atomically. This rule is needed in some cases for memory-mapped device registers. 19.c/5

Discussion: The atomic reads and writes associated with accesses to nonatomic components of a full access object that is atomic are normal atomic operations — all of the rules that apply to other atomic operations apply to these as well. In particular, these atomic reads and writes are sequential if they apply to the same object. 19.d/5

Implementation Requirements

20/5 {*AI12-0128-1*} {*AI12-0439-1*} The external effect of a program (see 1.1.3) is defined to include each read and update of a volatile or atomic object. The implementation shall not generate any memory reads or updates of atomic or volatile objects other than those specified by the program. However, there may be target-dependent cases where reading or writing a volatile but nonatomic object (typically a component) necessarily involves reading and/or writing neighboring storage, and that neighboring storage can overlap a volatile object.

20.a **Discussion:** The presumption is that volatile or atomic objects might reside in an "active" part of the address space where each read has a potential side effect, and at the very least might deliver a different value.

20.b The rule above and the definition of external effect are intended to prevent (at least) the following incorrect optimizations, where V is a volatile variable:

20.c - X:= V; Y:=V; cannot be allowed to be translated as Y:=V; X:=V;

20.d - Deleting redundant loads: X:= V; X:= V; shall read the value of V from memory twice.

20.e - Deleting redundant stores: V:= X; V:= X; shall write into V twice.

20.f - Extra stores: V:= X+Y; should not translate to something like V:= X; V:= V+Y;

20.g - Extra loads: X:= V; Y:= X+Z; X:=X+B; should not translate to something like Y:= V+Z; X:= V+B;

20.h - Reordering of loads from volatile variables: X:= V1; Y:= V2; (whether or not V1 = V2) should not translate to Y:= V2; X:= V1;

20.i - Reordering of stores to volatile variables: V1:= X; V2:= X; should not translate to V2:=X; V1:= X;

20.j/5 {*AI12-0128-1*} The part about "target-dependent cases" is intended to let compilers use a read-modify-write operation when a volatile component has a size that cannot be directly read or written with the available machine instructions. (For instance, writing a Boolean component with size 1 in a volatile packed array of Boolean requires a pre-read on most existing machines.)

21/4 *This paragraph was deleted.*{*AI05-0229-1*} {*AI12-0001-1*}

21.a/4 *This paragraph was deleted.*{*AI05-0009-1*} {*AI12-0001-1*}

Implementation Permissions

21.1/5 {*AI12-0282-1*} Within the body of an instance of a generic unit that has a formal type *T* that is not atomic and an actual type that is atomic, if an object *O* of type *T* is declared and explicitly specified as atomic, the implementation may introduce an additional copy on passing *O* to a subprogram with a parameter of type *T* that is normally passed by reference. A corresponding permission applies to volatile parameter passing.

Implementation Advice

22/5 {*AI95-00259-01*} {*AI12-0128-1*} A load or store of a volatile object whose size is a multiple of System.Storage_Unit and whose alignment is nonzero, should be implemented by accessing exactly the bits of the object and no others, except in the case of a volatile but nonatomic subcomponent of an atomic object.

22.a/2 **Implementation Advice:** A load or store of a volatile object whose size is a multiple of System.Storage_Unit and whose alignment is nonzero, should be implemented by accessing exactly the bits of the object and no others.

22.b/5 **Reason:** {*AI12-0128-1*} Since any object can be a volatile object, including packed array components and bit-mapped record components, we require the above only when it is reasonable to assume that the machine can avoid accessing bits outside of the object. The exception is needed so this advice doesn't conflict with other rules of this subclauase.

22.c/2 **Ramification:** This implies that the load or store of a volatile object that meets the above requirement should not be combined with that of any other object, nor should it access any bits not belonging to any other object. This means that the suitability of the implementation for memory-mapped I/O can be determined from its documentation, as any cases where the implementation does not follow Implementation Advice must be documented.

23/2 {*AI95-00259-01*} A load or store of an atomic object should, where possible, be implemented by a single load or store instruction.

Implementation Advice: A load or store of an atomic object should be implemented by a single load or store instruction. 23.a/2

NOTE 1 An imported volatile or atomic constant behaves as a constant (i.e. read-only) with respect to other parts of the Ada program, but can still be modified by an “external source”. 24

NOTE 2 {*AI12-0001-1*} Specifying the Pack aspect cannot override the effect of specifying an Atomic or Atomic_Components aspect. 25/4

NOTE 3 {*AI12-0128-1*} {*AI12-0440-1*} When mapping an Ada object to a memory-mapped hardware register, the Ada object can be declared atomic to ensure that the compiler will read and write exactly the bits of the register as specified in the source code and no others. 26/5

Discussion: This is especially important for a write-only hardware register, as in such a case a read-modify-write cycle will not work. The only time the language guarantees such a cycle will not happen is when writing an entire atomic object. If one wants to write individual components of a write-only hardware register (assuming the hardware supports that), those also need to be declared atomic. 26.a/5

Incompatibilities With Ada 83

Pragma Atomic replaces Ada 83's pragma Shared. The name “Shared” was confusing, because the pragma was not used to mark variables as shared. 26.b

Wording Changes from Ada 95

{*AI95-00259-01*} Added Implementation Advice to clarify the meaning of Atomic and Volatile in machine terms. The documentation that this advice applies will make the use of Ada implementations more predictable for low-level (such as device register) programming. 26.c/2

{*AI95-00272-01*} Added wording to clarify that a slice of an object of an atomic type is not atomic, just like a component of an atomic type is not (necessarily) atomic. 26.d/2

Incompatibilities With Ada 2005

{*AI05-0218-1*} **Correction:** Plugged a hole involving volatile components of formal types when the formal type's component has a nonvolatile type. This was done by making certain actual types illegal for formal derived and formal array types; these types were allowed for Ada 95 and Ada 2005. 26.e/3

Extensions to Ada 2005

{*AI05-0009-1*} {*AI05-0229-1*} Aspects Independent and Independent_Components are new; they eliminate ambiguity about independent addressability. 26.f/3

{*AI05-0229-1*} Aspects Atomic, Atomic_Components, Volatile, and Volatile_Components are new; pragmas Atomic, Atomic_Components, Volatile, and Volatile_Components are now obsolescent. 26.g/3

Wording Changes from Ada 2005

{*AI05-0117-1*} {*AI05-0275-1*} Revised the definition of volatile to eliminate overspecification and simply focus on the root requirement (that all tasks see the same view of volatile objects). This is not an inconsistency; "memory" arguably includes on-chip caches so long as those are kept consistent. Moreover, it is difficult to imagine a program that could tell the difference. 26.h/3

{*AI05-0142-4*} Added wording to take explicitly aliased parameters (see 6.1) into account when determining the legality of parameter passing of volatile and atomic objects. 26.i/3

Inconsistencies With Ada 2012

{*AI12-0128-1*} Required that nonatomic components of a atomic object use a read-modify-write cycle, as well as clarifying that such a cycle is allowed for volatile objects that aren't exactly the size of a machine scalar. That can introduce a read-modify-write cycle where one was not used previously (for instance, if the components are exactly byte-sized on most machines); it is thought that this will more often fix bugs with access to hardware than it will cause them. If this is an issue, declaring the components themselves as atomic will restore the previous behavior. 26.j/5

Extensions to Ada 2012

{*AI12-0282-1*} These aspects now can be specified for generic formal types. 26.k/5

{*AI12-0363-1*} Aspect Full_Access_Only is new; it can be used to guarantee access to a complete device register for any operation even when the register is mapped to a number of components. 26.l/5

Wording Changes from Ada 2012

26.m/4 {*AI12-0001-1*} **Corrigendum:** Clarified that aliased objects are considered to be specified as independently addressable, and also eliminated an unnecessary rule.

C.6.1 The Package System.Atomic_Operations

1/5 {*AI12-0234-1*} The language-defined package System.Atomic_Operations is the parent of a set of child units that provide facilities for manipulating objects of atomic types and for supporting lock-free synchronization. The subprograms of this subsystem are Intrinsic subprograms (see 6.3.1) in order to provide convenient access to machine operations that can provide these capabilities if they are available in the target environment.

Static Semantics

2/5 {*AI12-0234-1*} The library package System.Atomic_Operations has the following declaration:

3/5
```
package System.Atomic_Operations
   with Pure, Nonblocking is
end System.Atomic_Operations;
```

4/5 {*AI12-0234-1*} System.Atomic_Operations serves as the parent of other language-defined library units that manipulate atomic objects; its declaration is empty.

5/5 {*AI12-0234-1*} A call to a subprogram is said to be *lock-free* if the subprogram is guaranteed to return from the call while keeping the processor of the logical thread of control busy for the duration of the call.

6/5 {*AI12-0234-1*} In each child package, a function Is_Lock_Free(...) is provided to check whether the operations of the child package can all be provided lock-free for a given object. Is_Lock_Free returns True if operations defined in the child package are lock-free when applied to the object denoted by Item, and Is_Lock_Free returns False otherwise.

Extensions to Ada 2012

6.a/5 {*AI12-0234-1*} This package is new.

C.6.2 The Package System.Atomic_Operations.Exchange

1/5 {*AI12-0234-1*} The language-defined generic package System.Atomic_Operations.Exchange provides the following operations:

2/5 - To atomically compare the value of two atomic objects, and update the first atomic object with a desired value if both objects were found to be equal, or otherwise update the second object with the value of the first object.

3/5 - To atomically update the value of an atomic object, and then return the value that the atomic object had just prior to the update.

Static Semantics

4/5 {*AI12-0234-1*} The generic library package System.Atomic_Operations.Exchange has the following declaration:

5/5
```
generic
   type Atomic_Type is private with Atomic;
package System.Atomic_Operations.Exchange
   with Pure, Nonblocking is
```

6/5
```
   function Atomic_Exchange (Item  : aliased in out Atomic_Type;
                             Value : Atomic_Type) return Atomic_Type
      with Convention => Intrinsic;
```

```
   function Atomic_Compare_And_Exchange                                    7/5
     (Item    : aliased in out Atomic_Type;
      Prior   : aliased in out Atomic_Type;
      Desired : Atomic_Type) return Boolean
      with Convention => Intrinsic;
   function Is_Lock_Free (Item : aliased Atomic_Type) return Boolean       8/5
      with Convention => Intrinsic;
end System.Atomic_Operations.Exchange;                                     9/5
```

{*AI12-0234-1*} Atomic_Exchange atomically assigns the value of Value to Item, and returns the previous value of Item. 10/5

{*AI12-0234-1*} Atomic_Compare_And_Exchange first evaluates the value of Prior. Atomic_Compare_And_Exchange then performs the following steps as part of a single indivisible operation: 11/5

- evaluates the value of Item; 12/5
- compares the value of Item with the value of Prior; 13/5
- if equal, assigns Item the value of Desired; 14/5
- otherwise, makes no change to the value of Item. 15/5

{*AI12-0234-1*} After these steps, if the value of Item and Prior did not match, Prior is assigned the original value of Item, and the function returns False. Otherwise, Prior is unaffected and the function returns True. 16/5

Examples

{*AI12-0234-1*} *Example of a spin lock using Atomic_Exchange:* 17/5

```
type Atomic_Boolean is new Boolean with Atomic;                            18/5
package Exchange is new
   Atomic_Operations.Exchange (Atomic_Type => Atomic_Boolean);
Lock : aliased Atomic_Boolean := False;                                    19/5
...                                                                        20/5
begin -- Some critical section, trying to get the lock:                    21/5
   -- Obtain the lock                                                      22/5
   while Exchange.Atomic_Exchange (Item => Lock, Value => True) loop
      null;
   end loop;
   ... -- Do stuff                                                         23/5
   Lock := False; -- Release the lock                                      24/5
end;
```

Extensions to Ada 2012

{*AI12-0234-1*} This package is new. 24.a/5

C.6.3 The Package System.Atomic_Operations.Test_and_Set

{*AI12-0321-1*} The language-defined package System.Atomic_Operations.Test_And_Set provides an operation to atomically set and clear an atomic flag object. 1/5

Static Semantics

2/5 {*AI12-0321-1*} The library package System.Atomic_Operations.Test_And_Set has the following declaration:

3/5
```
package System.Atomic_Operations.Test_And_Set
   with Pure, Nonblocking is
```

4/5
```
   type Test_And_Set_Flag is mod implementation-defined
      with Atomic, Default_Value => 0, Size => implementation-defined;
```

5/5
```
   function Atomic_Test_And_Set
     (Item : aliased in out Test_And_Set_Flag) return Boolean
      with Convention => Intrinsic;
```

6/5
```
   procedure Atomic_Clear
     (Item : aliased in out Test_And_Set_Flag)
      with Convention => Intrinsic;
```

7/5
```
   function Is_Lock_Free
     (Item : aliased Test_And_Set_Flag) return Boolean
      with Convention => Intrinsic;
```

8/5
```
end System.Atomic_Operations.Test_And_Set;
```

8.a/5 **Implementation defined:** The modulus and size of Test_and_Set_Flag.

9/5 {*AI12-0321-1*} Test_And_Set_Flag represents the state of an atomic flag object. An atomic flag object can either be considered to be set or cleared.

10/5 {*AI12-0321-1*} Atomic_Test_And_Set performs an atomic test-and-set operation on Item. Item is set to some implementation-defined nonzero value. The function returns True if the previous contents were nonzero, and otherwise returns False.

10.a/5 **Implementation defined:** The value used to represent the set value for Atomic_Test_and_Set.

11/5 {*AI12-0321-1*} Atomic_Clear performs an atomic clear operation on Item. After the operation, Item contains 0. This call should be used in conjunction with Atomic_Test_And_Set.

Extensions to Ada 2012

11.a/5 {*AI12-0321-1*} This package is new.

C.6.4 The Package System.Atomic_Operations.Integer_Arithmetic

1/5 {*AI12-0321-1*} {*AI12-0364-1*} The language-defined generic package System.Atomic_Operations.Integer_Arithmetic provides operations to perform arithmetic atomically on objects of integer types.

Static Semantics

2/5 {*AI12-0321-1*} {*AI12-0364-1*} The generic library package System.Atomic_Operations.Integer_Arithmetic has the following declaration:

3/5
```
generic
   type Atomic_Type is range <> with Atomic;
package System.Atomic_Operations.Integer_Arithmetic
   with Pure, Nonblocking is
```

4/5
```
   procedure Atomic_Add (Item  : aliased in out Atomic_Type;
                         Value : Atomic_Type)
      with Convention => Intrinsic;
```

5/5
```
   procedure Atomic_Subtract (Item  : aliased in out Atomic_Type;
                              Value : Atomic_Type)
      with Convention => Intrinsic;
```

```
   function Atomic_Fetch_And_Add                                                    6/5
     (Item  : aliased in out Atomic_Type;
      Value : Atomic_Type) return Atomic_Type
      with Convention => Intrinsic;

   function Atomic_Fetch_And_Subtract                                               7/5
     (Item  : aliased in out Atomic_Type;
      Value : Atomic_Type) return Atomic_Type
      with Convention => Intrinsic;

   function Is_Lock_Free (Item : aliased Atomic_Type) return Boolean                8/5
      with Convention => Intrinsic;

end System.Atomic_Operations.Integer_Arithmetic;                                    9/5
```

{*AI12-0321-1*} The operations of this package are defined as follows: 10/5

```
procedure Atomic_Add (Item  : aliased in out Atomic_Type;                           11/5
                      Value : Atomic_Type)
   with Convention => Intrinsic;
```

{*AI12-0321-1*} Atomically performs: `Item := Item + Value;` 12/5

```
procedure Atomic_Subtract (Item  : aliased in out Atomic_Type;                      13/5
                           Value : Atomic_Type)
   with Convention => Intrinsic;
```

{*AI12-0321-1*} Atomically performs: `Item := Item - Value;` 14/5

```
function Atomic_Fetch_And_Add                                                       15/5
  (Item  : aliased in out Atomic_Type;
   Value : Atomic_Type) return Atomic_Type
   with Convention => Intrinsic;
```

{*AI12-0321-1*} Atomically performs: `Tmp := Item; Item := Item + Value; return Tmp;` 16/5

```
function Atomic_Fetch_And_Subtract                                                  17/5
  (Item  : aliased in out Atomic_Type;
   Value : Atomic_Type) return Atomic_Type
   with Convention => Intrinsic;
```

{*AI12-0321-1*} Atomically performs: `Tmp := Item; Item := Item - Value; return Tmp;` 18/5

Extensions to Ada 2012

{*AI12-0321-1*} This package is new. 18.a/5

C.6.5 The Package System.Atomic_Operations.Modular_Arithmetic

{*AI12-0364-1*} The language-defined generic package System.Atomic_Operations.Modular_Arithmetic provides operations to perform arithmetic atomically on objects of modular types. 1/5

Static Semantics

{*AI12-0364-1*} The generic library package System.Atomic_Operations.Modular_Arithmetic has the following declaration: 2/5

```
generic                                                                             3/5
   type Atomic_Type is mod <> with Atomic;
package System.Atomic_Operations.Modular_Arithmetic
   with Pure, Nonblocking is

   procedure Atomic_Add (Item  : aliased in out Atomic_Type;                        4/5
                         Value : Atomic_Type)
      with Convention => Intrinsic;
```

5/5
```
      procedure Atomic_Subtract (Item  : aliased in out Atomic_Type;
                                 Value : Atomic_Type)
         with Convention => Intrinsic;
```

6/5
```
      function Atomic_Fetch_And_Add
        (Item  : aliased in out Atomic_Type;
         Value : Atomic_Type) return Atomic_Type
         with Convention => Intrinsic;
```

7/5
```
      function Atomic_Fetch_And_Subtract
        (Item  : aliased in out Atomic_Type;
         Value : Atomic_Type) return Atomic_Type
         with Convention => Intrinsic;
```

8/5
```
      function Is_Lock_Free (Item : aliased Atomic_Type) return Boolean
         with Convention => Intrinsic;
```

9/5
```
   end System.Atomic_Operations.Modular_Arithmetic;
```

10/5 {*AI12-0364-1*} The operations of this package are defined as follows:

11/5
```
   procedure Atomic_Add (Item  : aliased in out Atomic_Type;
                         Value : Atomic_Type)
      with Convention => Intrinsic;
```

12/5 {*AI12-0364-1*} Atomically performs: `Item := Item + Value;`

13/5
```
   procedure Atomic_Subtract (Item  : aliased in out Atomic_Type;
                              Value : Atomic_Type)
      with Convention => Intrinsic;
```

14/5 {*AI12-0364-1*} Atomically performs: `Item := Item - Value;`

15/5
```
   function Atomic_Fetch_And_Add
     (Item  : aliased in out Atomic_Type;
      Value : Atomic_Type) return Atomic_Type
      with Convention => Intrinsic;
```

16/5 {*AI12-0364-1*} Atomically performs: `Tmp := Item; Item := Item + Value; return Tmp;`

17/5
```
   function Atomic_Fetch_And_Subtract
     (Item  : aliased in out Atomic_Type;
      Value : Atomic_Type) return Atomic_Type
      with Convention => Intrinsic;
```

18/5 {*AI12-0364-1*} Atomically performs: `Tmp := Item; Item := Item - Value; return Tmp;`

Extensions to Ada 2012

18.a/5 {*AI12-0364-1*} This package is new.

C.7 Task Information

1/3 {*AI95-00266-02*} {*AI05-0299-1*} [This subclause describes operations and attributes that can be used to obtain the identity of a task. In addition, a package that associates user-defined information with a task is defined. Finally, a package that associates termination procedures with a task or set of tasks is defined.]

Wording Changes from Ada 95

1.a/3 {*AI95-00266-02*} {*AI05-0299-1*} The title and text here were updated to reflect the addition of task termination procedures to this subclause.

C.7.1 The Package Task_Identification

Static Semantics

The following language-defined library package exists: 1

```
{AI95-00362-01} {AI12-0241-1} {AI12-0302-1} {AI12-0399-1} package                    2/5
Ada.Task_Identification
   with Preelaborate, Nonblocking, Global => in out synchronized is
   type Task_Id is private
      with Preelaborable_Initialization;
   Null_Task_Id : constant Task_Id;
   function  "=" (Left, Right : Task_Id) return Boolean;

{8652/0070} {AI95-00101-01} {AI05-0189-1} {AI12-0241-1}    function  Image          3/5
(T : Task_Id) return String;
   function  Current_Task     return Task_Id;
   function  Environment_Task return Task_Id;
   procedure Abort_Task             (T : in Task_Id)
      with Nonblocking => False;

{AI05-0189-1}    function  Is_Terminated          (T : Task_Id) return Boolean;      4/3
   function  Is_Callable            (T : Task_Id) return Boolean;
   function  Activation_Is_Complete (T : Task_Id) return Boolean;
private
   ... -- not specified by the language
end Ada.Task_Identification;
```

Dynamic Semantics

A value of the type Task_Id identifies an existent task. The constant Null_Task_Id does not identify any task. Each object of the type Task_Id is default initialized to the value of Null_Task_Id. 5

The function "=" returns True if and only if Left and Right identify the same task or both have the value Null_Task_Id. 6

The function Image returns an implementation-defined string that identifies T. If T equals Null_Task_Id, Image returns an empty string. 7

Implementation defined: The result of the Task_Identification.Image attribute. 7.a

The function Current_Task returns a value that identifies the calling task. 8

Ramification: {*AI12-0005-1*} The logical threads of control associated with the execution of a given parallel construct all execute as part of the execution of one task (see 9, “Tasks and Synchronization”). Thus, the result returned by a call to Task_Identification.Current_Task is independent of whether the call takes place during the execution of a parallel construct. 8.a.1/5

{*AI05-0189-1*} The function Environment_Task returns a value that identifies the environment task. 8.1/3

{*AI12-0454-1*} The effect of Abort_Task is the same as the abort_statement for the task identified by T. [In addition, if T identifies the environment task, the entire partition is aborted, see E.1.] 9/5

The functions Is_Terminated and Is_Callable return the value of the corresponding attribute of the task identified by T. 10

Ramification: {*8652/0115*} {*AI95-00206-01*} These routines can be called with an argument identifying the environment task. Is_Terminated will always be False for such a call, but Is_Callable (usually True) could be False if the environment task is waiting for the termination of dependent tasks. Thus, a dependent task can use Is_Callable to determine if the main subprogram has completed. 10.a.1/1

{*AI05-0189-1*} The function Activation_Is_Complete returns True if the task identified by T has completed its activation (whether successfully or not). It returns False otherwise. If T identifies the environment task, Activation_Is_Complete returns True after the elaboration of the library_items of the partition has completed. 10.1/3

11 For a prefix T that is of a task type [(after any implicit dereference)], the following attribute is defined:

12 T'Identity Yields a value of the type Task_Id that identifies the task denoted by T.

13 For a prefix E that denotes an entry_declaration, the following attribute is defined:

14/3 E'Caller {*AI05-0262-1*} Yields a value of the type Task_Id that identifies the task whose call is now being serviced. Use of this attribute is allowed only inside an accept_statement, or entry_body after the entry_barrier, corresponding to the entry_declaration denoted by E.

15/5 {*AI12-0231-1*} Program_Error is raised if a value of Null_Task_Id is passed as a parameter to Abort_Task, Activation_Is_Complete, Is_Terminated, and Is_Callable.

16/5 *This paragraph was deleted.*{*AI12-0241-1*}

Bounded (Run-Time) Errors

17/3 {*AI95-00237-01*} {*AI05-0004-1*} It is a bounded error to call the Current_Task function from an entry_body, interrupt handler, or finalization of a task attribute. Program_Error is raised, or an implementation-defined value of the type Task_Id is returned.

17.a/2 **Implementation defined:** The value of Current_Task when in a protected entry, interrupt handler, or finalization of a task attribute.

17.b **Implementation Note:** This value could be Null_Task_Id, or the ID of some user task, or that of an internal task created by the implementation.

17.c/2 **Ramification:** {*AI95-00237-01*} An entry barrier is syntactically part of an entry_body, so a call to Current_Task from an entry barrier is also covered by this rule.

Erroneous Execution

18 If a value of Task_Id is passed as a parameter to any of the operations declared in this package (or any language-defined child of this package), and the corresponding task object no longer exists, the execution of the program is erroneous.

Documentation Requirements

19 The implementation shall document the effect of calling Current_Task from an entry body or interrupt handler.

19.a/2 *This paragraph was deleted.*

19.b/2 **Documentation Requirement:** The effect of calling Current_Task from an entry body or interrupt handler.

20 NOTE 1 This package is intended for use in writing user-defined task scheduling packages and constructing server tasks. Current_Task can be used in conjunction with other operations requiring a task as an argument such as Set_Priority (see D.5).

21 NOTE 2 The function Current_Task and the attribute Caller can return a Task_Id value that identifies the environment task.

Extensions to Ada 95

21.a/2 {*AI95-00362-01*} Task_Identification is now preelaborated, so it can be used in preelaborated units.

Wording Changes from Ada 95

21.b/2 {*8652/0070*} {*AI95-00101-01*} **Corrigendum:** Corrected the mode of the parameter to Abort_Task to **in**.

21.c/2 {*AI95-00237-01*} Corrected the wording to include finalization of a task attribute in the bounded error case; we don't want to specify which task does these operations.

Incompatibilities With Ada 2005

21.d/3 {*AI05-0189-1*} Functions Environment_Task and Activation_Is_Complete are added to Task_Identification. If Task_Identification is referenced in a use_clause, and an entity *E* with a defining_identifier of Environment_Task or

Activation_Is_Complete is defined in a package that is also referenced in a use_clause, the entity *E* may no longer be use-visible, resulting in errors. This should be rare and is easily fixed if it does occur.

Wording Changes from Ada 2012

{*AI12-0231-1*} **Correction:** Defined what happens if Null_Task_Id is passed to Activation_Is_Complete. 21.e/5

C.7.2 The Package Task_Attributes

Static Semantics

The following language-defined generic library package exists: 1

```
{AI12-0241-1} {AI12-0302-1} with Ada.Task_Identification; use                    2/5
Ada.Task_Identification;
generic
   type Attribute is private;
   Initial_Value : in Attribute;
package Ada.Task_Attributes
   with Nonblocking, Global => in out synchronized is

   type Attribute_Handle is access all Attribute;                                3

   function Value(T : Task_Id := Current_Task)                                   4
     return Attribute;

   function Reference(T : Task_Id := Current_Task)                               5
     return Attribute_Handle;

   procedure Set_Value(Val : in Attribute;                                       6
                       T : in Task_Id := Current_Task);
   procedure Reinitialize(T : in Task_Id := Current_Task);

end Ada.Task_Attributes;                                                         7
```

Dynamic Semantics

When an instance of Task_Attributes is elaborated in a given active partition, an object of the actual type corresponding to the formal type Attribute is implicitly created for each task (of that partition) that exists and is not yet terminated. This object acts as a user-defined attribute of the task. A task created previously in the partition and not yet terminated has this attribute from that point on. Each task subsequently created in the partition will have this attribute when created. In all these cases, the initial value of the given attribute is Initial_Value. 8

The Value operation returns the value of the corresponding attribute of T. 9

The Reference operation returns an access value that designates the corresponding attribute of T. 10

The Set_Value operation performs any finalization on the old value of the attribute of T and assigns Val to that attribute (see 5.2 and 7.6). 11

The effect of the Reinitialize operation is the same as Set_Value where the Val parameter is replaced with Initial_Value. 12

Implementation Note: In most cases, the attribute memory can be reclaimed at this point. 12.a

For all the operations declared in this package, Tasking_Error is raised if the task identified by T is terminated. Program_Error is raised if the value of T is Null_Task_Id. 13

{*AI95-00237-01*} After a task has terminated, all of its attributes are finalized, unless they have been finalized earlier. When the master of an instantiation of Ada.Task_Attributes is finalized, the corresponding attribute of each task is finalized, unless it has been finalized earlier. 13.1/2

Reason: This is necessary so that a task attribute does not outlive its type. For instance, that's possible if the instantiation is nested, and the attribute is on a library-level task. 13.a/2

13.b/2 **Ramification:** The task owning an attribute cannot, in general, finalize that attribute. That's because the attributes are finalized *after* the task is terminated; moreover, a task may have attributes as soon as it is created; the task may never even have been activated.

Bounded (Run-Time) Errors

13.2/1 {*8652/0071*} {*AI95-00165-01*} If the package Ada.Task_Attributes is instantiated with a controlled type and the controlled type has user-defined Adjust or Finalize operations that in turn access task attributes by any of the above operations, then a call of Set_Value of the instantiated package constitutes a bounded error. The call may perform as expected or may result in forever blocking the calling task and subsequently some or all tasks of the partition.

Erroneous Execution

14 It is erroneous to dereference the access value returned by a given call on Reference after a subsequent call on Reinitialize for the same task attribute, or after the associated task terminates.

14.a **Reason:** This allows the storage to be reclaimed for the object associated with an attribute upon Reinitialize or task termination.

15 If a value of Task_Id is passed as a parameter to any of the operations declared in this package and the corresponding task object no longer exists, the execution of the program is erroneous.

15.1/2 {*8652/0071*} {*AI95-00165-01*} {*AI95-00237-01*} An access to a task attribute via a value of type Attribute_Handle is erroneous if executed concurrently with another such access or a call of any of the operations declared in package Task_Attributes. An access to a task attribute is erroneous if executed concurrently with or after the finalization of the task attribute.

15.a.1/1 **Reason:** There is no requirement of atomicity on accesses via a value of type Attribute_Handle.

15.a.2/2 **Ramification:** A task attribute can only be accessed after finalization through a value of type Attribute_Handle. Operations in package Task_Attributes cannot be used to access a task attribute after finalization, because either the master of the instance has been or is in the process of being left (in which case the instance is out of scope and thus cannot be called), or the associated task is already terminated (in which case Tasking_Error is raised for any attempt to call a task attribute operation).

Implementation Requirements

16/1 {*8652/0071*} {*AI95-00165-01*} For a given attribute of a given task, the implementation shall perform the operations declared in this package atomically with respect to any of these operations of the same attribute of the same task. The granularity of any locking mechanism necessary to achieve such atomicity is implementation defined.

16.a.1/1 **Implementation defined:** Granularity of locking for Task_Attributes.

16.a **Ramification:** Hence, other than by dereferencing an access value returned by Reference, an attribute of a given task can be safely read and updated concurrently by multiple tasks.

17/2 {*AI95-00237-01*} After task attributes are finalized, the implementation shall reclaim any storage associated with the attributes.

Documentation Requirements

18 The implementation shall document the limit on the number of attributes per task, if any, and the limit on the total storage for attribute values per task, if such a limit exists.

19 In addition, if these limits can be configured, the implementation shall document how to configure them.

19.a/2 *This paragraph was deleted.*

19.b/2 **Documentation Requirement:** For package Task_Attributes, limits on the number and size of task attributes, and how to configure any limits.

Metrics

{*AI95-00434-01*} The implementation shall document the following metrics: A task calling the following subprograms shall execute at a sufficiently high priority as to not be preempted during the measurement period. This period shall start just before issuing the call and end just after the call completes. If the attributes of task T are accessed by the measurement tests, no other task shall access attributes of that task during the measurement period. For all measurements described here, the Attribute type shall be a scalar type whose size is equal to the size of the predefined type Integer. For each measurement, two cases shall be documented: one where the accessed attributes are of the calling task [(that is, the default value for the T parameter is used)], and the other, where T identifies another, nonterminated, task. 20/2

The following calls (to subprograms in the Task_Attributes package) shall be measured: 21

- a call to Value, where the return value is Initial_Value; 22
- a call to Value, where the return value is not equal to Initial_Value; 23
- a call to Reference, where the return value designates a value equal to Initial_Value; 24
- a call to Reference, where the return value designates a value not equal to Initial_Value; 25
- {*AI95-00434-01*} a call to Set_Value where the Val parameter is not equal to Initial_Value and the old attribute value is equal to Initial_Value; 26/2
- a call to Set_Value where the Val parameter is not equal to Initial_Value and the old attribute value is not equal to Initial_Value. 27

Documentation Requirement: The metrics for the Task_Attributes package. 27.a/2

Implementation Permissions

{*AI12-0444-1*} An implementation can avoid actually creating the object corresponding to a task attribute until its value is set to something other than that of Initial_Value, or until Reference is called for the task attribute. Similarly, when the value of the attribute is to be reinitialized to that of Initial_Value, the object may instead be finalized and its storage reclaimed, to be recreated when needed later. While the object does not exist, the function Value may simply return Initial_Value, rather than implicitly creating the object. 28/5

Discussion: The effect of this permission can only be observed if the assignment operation for the corresponding type has side effects. 28.a

Implementation Note: {*AI95-00114-01*} This permission means that even though every task has every attribute, storage need only be allocated for those attributes for which function Reference has been invoked or set to a value other than that of Initial_Value. 28.b/2

An implementation is allowed to place restrictions on the maximum number of attributes a task may have, the maximum size of each attribute, and the total storage size allocated for all the attributes of a task. 29

Implementation Advice

{*AI95-00434-01*} {*AI12-0438-1*} Some implementations are targeted to domains in which memory use at run time has to be completely deterministic. For such implementations, it is recommended that the storage for task attributes will be pre-allocated statically and not from the heap. This can be accomplished by either placing restrictions on the number and the size of the attributes of a task, or by using the pre-allocated storage for the first N attribute objects, and the heap for the others. In the latter case, N should be documented. 30/5

Implementation Advice: If the target domain requires deterministic memory use at run time, storage for task attributes should be pre-allocated statically and the number of attributes pre-allocated should be documented. 30.a/2

Discussion: We don't mention “restrictions on the size and number” (that is, limits) in the text for the Annex, because it is covered by the Documentation Requirement above, and we try not to repeat requirements in the Annex (they're enough work to meet without having to do things twice). 30.b/2

30.1/2 {*AI95-00237-01*} Finalization of task attributes and reclamation of associated storage should be performed as soon as possible after task termination.

30.c/2 **Implementation Advice:** Finalization of task attributes and reclamation of associated storage should be performed as soon as possible after task termination.

30.d/2 **Reason:** {*AI95-00237-01*} This is necessary because the normative wording only says that attributes are finalized "after" task termination. Without this advice, waiting until the instance is finalized would meet the requirements (it is after termination, but may be a very long time after termination). We can't say anything more specific than this, as we do not want to require the overhead of an interaction with the tasking system to be done at a specific point.

31/5 NOTE 1 {*AI12-0442-1*} An attribute always exists (after instantiation), and has the initial value. An implementation can avoid using memory to store the attribute value until the first operation that changes the attribute value. The same holds true after Reinitialize.

32/5 NOTE 2 {*AI12-0442-1*} The result of the Reference function is always safe to use in the task body whose attribute is being accessed. However, when the result is being used by another task, the programmer will want to make sure that the task whose attribute is being accessed is not yet terminated. Failing to do so can make the program execution erroneous.

Wording Changes from Ada 95

33.a/2 {*8652/0071*} {*AI95-00165-01*} **Corrigendum:** Clarified that use of task attribute operations from within a task attribute operation (by an Adjust or Finalize call) is a bounded error, and that concurrent use of attribute handles is erroneous.

33.b/2 {*AI95-00237-01*} Clarified the wording so that the finalization takes place after the termination of the task or when the instance is finalized (whichever is sooner).

C.7.3 The Package Task_Termination

Static Semantics

1/2 {*AI95-00266-02*} The following language-defined library package exists:

2/5
```
{AI12-0241-1} {AI12-0302-1} with Ada.Task_Identification;
with Ada.Exceptions;
package Ada.Task_Termination
   with Preelaborate, Nonblocking, Global => in out synchronized is

3/2    type Cause_Of_Termination is (Normal, Abnormal, Unhandled_Exception);

4/2    type Termination_Handler is access protected procedure
         (Cause : in Cause_Of_Termination;
          T     : in Ada.Task_Identification.Task_Id;
          X     : in Ada.Exceptions.Exception_Occurrence);

5/2    procedure Set_Dependents_Fallback_Handler
         (Handler: in Termination_Handler);
       function Current_Task_Fallback_Handler return Termination_Handler;

6/2    procedure Set_Specific_Handler
         (T       : in Ada.Task_Identification.Task_Id;
          Handler : in Termination_Handler);
       function Specific_Handler (T : Ada.Task_Identification.Task_Id)
          return Termination_Handler;

7/2 end Ada.Task_Termination;
```

Dynamic Semantics

8/3 {*AI95-00266-02*} {*AI05-0202-1*} The type Termination_Handler identifies a protected procedure to be executed by the implementation when a task terminates. Such a protected procedure is called a *handler*. In all cases T identifies the task that is terminating. If the task terminates due to completing the last statement of its body, or as a result of waiting on a terminate alternative, and the finalization of the task completes normally, then Cause is set to Normal and X is set to Null_Occurrence. If the task terminates because it is being aborted, then Cause is set to Abnormal; X is set to Null_Occurrence if the finalization of the task completes normally. If the task terminates because of an exception raised by the execution of its task_body, then Cause is set to Unhandled_Exception; X is set to the associated exception occurrence

if the finalization of the task completes normally. Independent of how the task completes, if finalization of the task propagates an exception, then Cause is either Unhandled_Exception or Abnormal, and X is an exception occurrence that identifies the Program_Error exception.

{*AI95-00266-02*} Each task has two termination handlers, a *fall-back handler* and a *specific handler*. The specific handler applies only to the task itself, while the fall-back handler applies only to the dependent tasks of the task. A handler is said to be *set* if it is associated with a nonnull value of type Termination_Handler, and *cleared* otherwise. When a task is created, its specific handler and fall-back handler are cleared. 9/2

{*AI95-00266-02*} {*AI05-0264-1*} The procedure Set_Dependents_Fallback_Handler changes the fall-back handler for the calling task: if Handler is **null**, that fall-back handler is cleared; otherwise, it is set to be Handler.**all**. If a fall-back handler had previously been set it is replaced. 10/3

{*AI95-00266-02*} {*AI05-0264-1*} The function Current_Task_Fallback_Handler returns the fall-back handler that is currently set for the calling task, if one is set; otherwise, it returns **null**. 11/3

{*AI95-00266-02*} {*AI05-0264-1*} The procedure Set_Specific_Handler changes the specific handler for the task identified by T: if Handler is **null**, that specific handler is cleared; otherwise, it is set to be Handler.**all**. If a specific handler had previously been set it is replaced. 12/3

Ramification: {*AI05-0005-1*} This package cannot portably be used to set a handler on the program as a whole. It is possible to call Set_Specific_Handler with the environment task's ID. But any call to the handler would necessarily be a Bounded (Run-Time) Error, as the handler is called after the task's finalization has completed. In the case of the environment task, that includes any possible protected objects, and calling a protected object after it is finalized is a Bounded (Run-Time) Error (see 9.4). This might work in a particular implementation, but it cannot be depended upon. 12.a/3

{*AI95-00266-02*} {*AI05-0264-1*} The function Specific_Handler returns the specific handler that is currently set for the task identified by T, if one is set; otherwise, it returns **null**. 13/3

{*AI95-00266-02*} As part of the finalization of a task_body, after performing the actions specified in 7.6 for finalization of a master, the specific handler for the task, if one is set, is executed. If the specific handler is cleared, a search for a fall-back handler proceeds by recursively following the master relationship for the task. If a task is found whose fall-back handler is set, that handler is executed; otherwise, no handler is executed. 14/2

{*AI95-00266-02*} For Set_Specific_Handler or Specific_Handler, Tasking_Error is raised if the task identified by T has already terminated. Program_Error is raised if the value of T is Ada.Task_Identification.Null_Task_Id. 15/2

{*AI95-00266-02*} An exception propagated from a handler that is invoked as part of the termination of a task has no effect. 16/2

Erroneous Execution

{*AI95-00266-02*} For a call of Set_Specific_Handler or Specific_Handler, if the task identified by T no longer exists, the execution of the program is erroneous. 17/2

Extensions to Ada 95

{*AI95-00266-02*} Package Task_Termination is new. 17.a/2

Wording Changes from Ada 2005

{*AI05-0202-1*} **Correction:** Specified what is passed to the handler if the finalization of the task fails after it is completed. This was not specified at all in Ada 2005, so there is a possibility that some program depended on some other behavior of an implementation. But as this case is very unlikely (and only occurs when there is already a significant bug in the program - so should not occur in fielded systems), we're not listing this as an inconsistency. 17.b/3

Annex D
(normative)
Real-Time Systems

{*AI12-0404-1*} This Annex specifies additional characteristics of Ada implementations intended for real-time systems software. To conform to this Annex, an implementation shall also conform to Annex C, "Systems Programming". 1/5

Metrics

The metrics are documentation requirements; an implementation shall document the values of the language-defined metrics for at least one configuration [of hardware or an underlying system] supported by the implementation, and shall document the details of that configuration. 2

This paragraph was deleted. 2.a/2

Documentation Requirement: The details of the configuration used to generate the values of all metrics. 2.a.1/2

Reason: The actual values of the metrics are likely to depend on hardware configuration details that are variable and generally outside the control of a compiler vendor. 2.b

The metrics do not necessarily yield a simple number. [For some, a range is more suitable, for others a formula dependent on some parameter is appropriate, and for others, it may be more suitable to break the metric into several cases.] Unless specified otherwise, the metrics in this annex are expressed in processor clock cycles. For metrics that require documentation of an upper bound, if there is no upper bound, the implementation shall report that the metric is unbounded. 3

Discussion: There are several good reasons to specify metrics in seconds; there are however equally good reasons to specify them in processor clock cycles. In defining the metrics, we have tried to strike a balance on a case-by-case basis. 3.a

It has been suggested that all metrics should be given names, so that "data-sheets" could be formulated and published by vendors. However the paragraph number can serve that purpose. 3.b

NOTE 1 The specification of the metrics makes a distinction between upper bounds and simple execution times. Where something is just specified as "the execution time of" a piece of code, this leaves one the freedom to choose a nonpathological case. This kind of metric is of the form "there exists a program such that the value of the metric is V". Conversely, the meaning of upper bounds is "there is no program such that the value of the metric is greater than V". This kind of metric can only be partially tested, by finding the value of V for one or more test programs. 4

NOTE 2 The metrics do not cover the whole language; they are limited to features that are specified in Annex C, "Systems Programming" and in this Annex. The metrics are intended to provide guidance to potential users as to whether a particular implementation of such a feature is going to be adequate for a particular real-time application. As such, the metrics are aimed at known implementation choices that can result in significant performance differences. 5

NOTE 3 The purpose of the metrics is not necessarily to provide fine-grained quantitative results or to serve as a comparison between different implementations on the same or different platforms. Instead, their goal is rather qualitative; to define a standard set of approximate values that can be measured and used to estimate the general suitability of an implementation, or to evaluate the comparative utility of certain features of an implementation for a particular real-time application. 6

Extensions to Ada 83

This Annex is new to Ada 95. 6.a

D.1 Task Priorities

{*AI05-0299-1*} [This subclause specifies the priority model for real-time systems. In addition, the methods for specifying priorities are defined.] 1/3

Paragraphs 2 through 6 were moved to Annex J, "Obsolescent Features".

Static Semantics

6.1/3 {*AI05-0229-1*} For a task type (including the anonymous type of a single_task_declaration), protected type (including the anonymous type of a single_protected_declaration), or subprogram, the following language-defined representation aspects may be specified:

6.2/3 Priority The aspect Priority is an expression, which shall be of type Integer.

6.a/3 **Aspect Description for Priority:** Priority of a task object or type, or priority of a protected object or type; the priority is not in the interrupt range.

6.3/3 Interrupt_Priority
The aspect Interrupt_Priority is an expression, which shall be of type Integer.

6.b/3 **Aspect Description for Interrupt_Priority:** Priority of a task object or type, or priority of a protected object or type; the priority is in the interrupt range.

Legality Rules

7/3 *This paragraph was deleted.*{*AI05-0229-1*}

8/3 {*AI05-0229-1*} If the Priority aspect is specified for a subprogram, the expression shall be static, and its value shall be in the range of System.Priority.

8.a **Reason:** This value is needed before it gets elaborated, when the environment task starts executing.

8.1/3 {*AI05-0229-1*} At most one of the Priority and Interrupt_Priority aspects may be specified for a given entity.

8.b/3 **Ramification:** This includes specifying via pragmas (see J.15.11). Note that 13.1 prevents multiple specifications of a single representation aspect by any means.

8.2/3 {*AI05-0229-1*} Neither of the Priority or Interrupt_Priority aspects shall be specified for a synchronized interface type.

Static Semantics

9 The following declarations exist in package System:

10
```
subtype Any_Priority is Integer range implementation-defined;
subtype Priority is Any_Priority
   range Any_Priority'First .. implementation-defined;
subtype Interrupt_Priority is Any_Priority
   range Priority'Last+1 .. Any_Priority'Last;
```

11
```
Default_Priority : constant Priority := (Priority'First + Priority'Last)/2;
```

11.a **Implementation defined:** The declarations of Any_Priority and Priority.

12 The full range of priority values supported by an implementation is specified by the subtype Any_Priority. The subrange of priority values that are high enough to require the blocking of one or more interrupts is specified by the subtype Interrupt_Priority. [The subrange of priority values below System.-Interrupt_Priority'First is specified by the subtype System.Priority.]

13/3 *This paragraph was deleted.*{*AI05-0229-1*}

Dynamic Semantics

14/3 {*AI05-0229-1*} The Priority aspect has no effect if it is specified for a subprogram other than the main subprogram; the Priority value is not associated with any task.

15/5 {*AI12-0404-1*} A *task priority* is an integer value that indicates a degree of urgency and is the basis for resolving competing demands of tasks for resources. Unless otherwise specified, whenever tasks compete for processors or other implementation-defined resources, the resources are allocated to the task with the highest priority value. The *base priority* of a task is the priority with which it was created, or to which it was later set by Dynamic_Priorities.Set_Priority (see D.5). At all times, a task also has an *active priority*,

which generally is its base priority unless it inherits a priority from other sources. *Priority inheritance* is the process by which the priority of a task or other entity (for example, a protected object; see D.3) is used in the evaluation of another task's active priority.

Implementation defined: Implementation-defined execution resources. 15.a

{*AI05-0229-1*} The effect of specifying a Priority or Interrupt_Priority aspect for a protected type or single_protected_declaration is discussed in D.3. 16/3

{*AI05-0229-1*} {*AI12-0081-1*} The expression specified for the Priority or Interrupt_Priority aspect of a task type is evaluated each time an object of the task type is created (see 9.1). For the Priority aspect, the value of the expression is converted to the subtype Priority; for the Interrupt_Priority aspect, this value is converted to the subtype Any_Priority. The priority value is then associated with the task object. 17/4

{*AI05-0229-1*} Likewise, the priority value is associated with the environment task if the aspect is specified for the main subprogram. 18/3

{*AI05-0229-1*} The initial value of a task's base priority is specified by default or by means of a Priority or Interrupt_Priority aspect. [After a task is created, its base priority can be changed only by a call to Dynamic_Priorities.Set_Priority (see D.5).] The initial base priority of a task in the absence of an aspect is the base priority of the task that creates it at the time of creation (see 9.1). If the aspect Priority is not specified for the main subprogram, the initial base priority of the environment task is System.Default_Priority. [The task's active priority is used when the task competes for processors. Similarly, the task's active priority is used to determine the task's position in any queue when Priority_Queuing is specified (see D.4).] 19/3

{*AI95-00357-01*} At any time, the active priority of a task is the maximum of all the priorities the task is inheriting at that instant. For a task that is not held (see D.11), its base priority is a source of priority inheritance unless otherwise specified for a particular task dispatching policy. Other sources of priority inheritance are specified under the following conditions: 20/2

Discussion: Other parts of the annex, e.g. D.11, define other sources of priority inheritance. 20.a

- {*8652/0072*} {*AI95-00092-01*} During activation, a task being activated inherits the active priority that its activator (see 9.2) had at the time the activation was initiated. 21/1
- {*8652/0072*} {*AI95-00092-01*} During rendezvous, the task accepting the entry call inherits the priority of the entry call (see 9.5.3 and D.4). 22/1
- {*AI12-0276-1*} While starting a protected action on a protected object when the FIFO_Spinning admission policy is in effect, a task inherits the ceiling priority of the protected object (see 9.5, D.3, and D.4.1). 22.1/5

  Reason: {*AI12-0005-1*} {*AI12-0276-1*} Priority inheritance is needed for FIFO_Spinning to ensure that lower priority tasks that initiate spin waiting earlier than other higher priority tasks continue to spin to ensure that they can be granted the resource when it becomes available in order to support FIFO ordering. Note that this rule only matters when tasks that can initiate a protected action on an object P can be on a different processor than P. In particular, this rule does not matter on a monoprocessor. 22.a/5
- {*AI12-0404-1*} While a task executes a protected action on a protected object, the task inherits the ceiling priority of the protected object (see 9.5 and D.3). 23/5

In all of these cases, the priority ceases to be inherited as soon as the condition calling for the inheritance no longer exists. 24

Implementation Requirements

The range of System.Interrupt_Priority shall include at least one value. 25

The range of System.Priority shall include at least 30 values. 26

27 NOTE 1 The priority expression can include references to discriminants of the enclosing type.

28 NOTE 2 It is a consequence of the active priority rules that at the point when a task stops inheriting a priority from another source, its active priority is re-evaluated. This is in addition to other instances described in this Annex for such re-evaluation.

29/5 NOTE 3 {*AI05-0248-1*} {*AI12-0440-1*} An implementation can provide a nonstandard mode in which tasks inherit priorities under conditions other than those specified above.

29.a/3 **Ramification:** {*AI05-0229-1*} The use of a Priority or Interrupt_Priority aspect does not require the package System to be named in a with_clause for the enclosing compilation_unit.

Extensions to Ada 83

29.b The priority of a task is per-object and not per-type.

29.c Priorities need not be static anymore (except for the main subprogram).

Wording Changes from Ada 83

29.d The description of the Priority pragma has been moved to this annex.

Wording Changes from Ada 95

29.e/2 {*8652/0072*} {*AI95-00092-01*} **Corrigendum:** Clarified that dynamic priority changes are not transitive - that is, they don't apply to tasks that are being activated by or in rendezvous with the task that had its priority changed.

29.f/2 {*AI95-00357-01*} Generalized the definition of priority inheritance to take into account the differences between the existing and new dispatching policies.

Extensions to Ada 2005

29.g/3 {*AI05-0229-1*} Aspects Priority and Interrupt_Priority are new; pragmas Priority and Interrupt_Priority are now obsolescent.

Wording Changes from Ada 2012

29.h/4 {*AI12-0081-1*} **Corrigendum:** Clarified when the Priority and Interrupt_Priority aspect expressions are evaluated.

29.i/5 {*AI12-0276-1*} Added an additional case of priority inheritance when the new admission policy FIFO_Spinning is in effect.

D.2 Priority Scheduling

1/3 {*AI95-00321-01*} {*AI05-0299-1*} [This subclause describes the rules that determine which task is selected for execution when more than one task is ready (see 9).]

Wording Changes from Ada 95

1.a/3 {*AI95-00321-01*} {*AI05-0299-1*} This introduction is simplified in order to reflect the rearrangement and expansion of this subclause.

D.2.1 The Task Dispatching Model

1/2 {*AI95-00321-01*} [The task dispatching model specifies task scheduling, based on conceptual priority-ordered ready queues.]

Static Semantics

1.1/2 {*AI95-00355-01*} The following language-defined library package exists:

1.2/5
```
{AI05-0166-1} {AI12-0241-1} {AI12-0302-1} package Ada.Dispatching
  with Preelaborate, Nonblocking, Global => in out synchronized is
```

1.3/5
```
   {AI05-0166-1} {AI12-0241-1}   procedure Yield
      with Nonblocking => False;
```

1.4/3
```
   {AI05-0166-1}   Dispatching_Policy_Error : exception;
end Ada.Dispatching;
```

Dispatching serves as the parent of other language-defined library units concerned with task dispatching. 1.5/2

{*AI12-0279-1*} For a noninstance subprogram (including a generic formal subprogram), a generic subprogram, or an entry, the following language-defined aspect may be specified with an aspect_specification (see 13.1.1): 1.6/5

Yield The type of aspect Yield is Boolean. 1.7/5

Aspect Description for Yield: Ensures that a callable entity includes a task dispatching point. 1.a/5

If directly specified, the aspect_definition shall be a static expression. If not specified (including by inheritance), the aspect is False. 1.8/5

{*AI12-0279-1*} {*AI12-0294-1*} If a Yield aspect is specified True for a primitive subprogram *S* of a type *T*, then the aspect is inherited by the corresponding primitive subprogram of each descendant of *T*. 1.9/5

Legality Rules

{*AI12-0279-1*} {*AI12-0294-1*} If the Yield aspect is specified for a dispatching subprogram that inherits the aspect, the specified value shall be confirming. 1.10/5

{*AI12-0279-1*} {*AI12-0294-1*} If the Nonblocking aspect (see 9.5) of the associated callable entity is statically True, the Yield aspect shall not be specified as True. For a callable entity that is declared within a generic body, this rule is checked assuming that any nonstatic Nonblocking attributes in the expression of the Nonblocking aspect of the entity are statically True. 1.11/5

Reason: {*AI12-0294-1*} The second sentence here is an assume-the-worst rule. The only Nonblocking attributes that are nonstatic are those that depend, directly or indirectly, on the nonblocking aspect of a generic formal parameter. We have to assume these might in fact have the value True if given an appropriate actual entity. 1.b/5

{*AI12-0294-1*} In addition to the places where Legality Rules normally apply (see 12.3), these rules also apply in the private part of an instance of a generic unit. 1.12/5

Dynamic Semantics

{*AI95-00321-01*} {*AI12-0454-1*} A task can become a *running task* only if it is ready (see Clause 9) and the execution resources required by that task are available. Processors are allocated to tasks based on each task's active priority. 2/5

It is implementation defined whether, on a multiprocessor, a task that is waiting for access to a protected object keeps its processor busy. 3

Implementation defined: Whether, on a multiprocessor, a task that is waiting for access to a protected object keeps its processor busy. 3.a

{*AI95-00321-01*} {*AI12-0119-1*} *Task dispatching* is the process by which a logical thread of control associated with a ready task is selected for execution on a processor. This selection is done during the execution of such a logical thread of control, at certain points called *task dispatching points*. Such a logical thread of control reaches a task dispatching point whenever it becomes blocked, and when its associated task terminates. [Other task dispatching points are defined throughout this Annex for specific policies.] Below we talk in terms of tasks, but in the context of a parallel construct, a single task can be represented by multiple logical threads of control, each of which can appear separately on a ready queue. 4/5

Ramification: On multiprocessor systems, more than one task can be chosen, at the same time, for execution on more than one processor, as explained below. 4.a

{*AI95-00321-01*} *Task dispatching policies* are specified in terms of conceptual *ready queues* and task states. A ready queue is an ordered list of ready tasks. The first position in a queue is called the *head of the queue*, and the last position is called the *tail of the queue*. A task is *ready* if it is in a ready queue, or if it is running. Each processor has one ready queue for each priority value. At any instant, each ready 5/2

queue of a processor contains exactly the set of tasks of that priority that are ready for execution on that processor, but are not running on any processor; that is, those tasks that are ready, are not running on any processor, and can be executed using that processor and other available resources. A task can be on the ready queues of more than one processor.

5.a **Discussion:** The core language defines a ready task as one that is not blocked. Here we refine this definition and talk about ready queues.

6/2 {*AI95-00321-01*} Each processor also has one *running task*, which is the task currently being executed by that processor. Whenever a task running on a processor reaches a task dispatching point it goes back to one or more ready queues; a task (possibly the same task) is then selected to run on that processor. The task selected is the one at the head of the highest priority nonempty ready queue; this task is then removed from all ready queues to which it belongs.

6.a **Discussion:** There is always at least one task to run, if we count the idle task.

7/5 {*AI95-00321-01*} {*AI05-0166-1*} {*AI12-0241-1*} A call of Yield and a delay_statement are task dispatching points for all language-defined policies.

7.a/2 *This paragraph was deleted.*

8/5 {*AI95-00321-01*} {*AI12-0279-1*} If the Yield aspect has the value True, then a call to procedure Yield is included within the body of the associated callable entity, and invoked immediately prior to returning from the body if and only if no other task dispatching points were encountered during the execution of the body.

8.a/2 *This paragraph was deleted.*

Implementation Permissions

9/2 {*AI95-00321-01*} An implementation is allowed to define additional resources as execution resources, and to define the corresponding allocation policies for them. Such resources may have an implementation-defined effect on task dispatching.

9.a/2 **Implementation defined:** The effect of implementation-defined execution resources on task dispatching.

10 An implementation may place implementation-defined restrictions on tasks whose active priority is in the Interrupt_Priority range.

10.a/3 **Ramification:** {*AI05-0229-1*} For example, on some operating systems, it might be necessary to disallow them altogether. This permission applies to tasks whose priority is set to interrupt level for any reason: via an aspect, via a call to Dynamic_Priorities.Set_Priority, or via priority inheritance.

10.1/5 {*AI95-00321-01*} {*AI12-0299-1*} Unless otherwise specified for a task dispatching policy, an implementation may add additional points at which task dispatching may occur, in an implementation-defined manner.

10.b/5 **Reason:** {*AI12-0299-1*} This permission is intended to allow the implementation of Ada tasks in terms of target system threads, which may have additional conditions that cause task dispatching. For instance, for Linux threads, page faults are task dispatching points.

10.c/5 **Discussion:** {*AI12-0299-1*} The Non_Preemptive_FIFO_Within_Priorities task dispatching policy (see D.2.4) does not allow additional task dispatching points.

11 NOTE 1 Clause 9 specifies under which circumstances a task becomes ready. The ready state is affected by the rules for task activation and termination, delay statements, and entry calls. When a task is not ready, it is said to be blocked.

12/5 NOTE 2 {*AI12-0442-1*} An example of a possible implementation-defined execution resource is a page of physical memory, which must be loaded with a particular page of virtual memory before a task can continue execution.

13 NOTE 3 The ready queues are purely conceptual; there is no requirement that such lists physically exist in an implementation.

14 NOTE 4 While a task is running, it is not on any ready queue. Any time the task that is running on a processor is added to a ready queue, a new running task is selected for that processor.

NOTE 5 In a multiprocessor system, a task can be on the ready queues of more than one processor. At the extreme, if several processors share the same set of ready tasks, the contents of their ready queues is identical, and so they can be viewed as sharing one ready queue, and can be implemented that way. [Thus, the dispatching model covers multiprocessors where dispatching is implemented using a single ready queue, as well as those with separate dispatching domains.] 15

NOTE 6 The priority of a task is determined by rules specified in this subclause, and under D.1, “Task Priorities”, D.3, “Priority Ceiling Locking”, and D.5, “Dynamic Priorities”. 16

NOTE 7 {*AI95-00321-01*} The setting of a task's base priority as a result of a call to Set_Priority does not always take effect immediately when Set_Priority is called. The effect of setting the task's base priority is deferred while the affected task performs a protected action. 17/2

Wording Changes from Ada 95

{*AI95-00321-01*} {*AI05-0005-1*} This description is simplified to describe only the parts of the dispatching model common to all policies. In particular, rules about preemption are moved elsewhere. This makes it easier to add other policies (which might not involve preemption). 17.a/3

Incompatibilities With Ada 2005

{*AI05-0166-1*} Procedure Yield is added to Dispatching. If Dispatching is referenced in a use_clause, and an entity *E* with a defining_identifier of Yield is defined in a package that is also referenced in a use_clause, the entity *E* may no longer be use-visible, resulting in errors. This should be rare and is easily fixed if it does occur. 17.b/3

{*AI05-0166-1*} {*AI12-0005-1*} Package Dispatching was a Pure package, but now is Preelaborated with the addition of Yield. This is incompatible as Dispatching can no longer be depended upon from a Pure package. This should happen rarely in practice as the only contents was the exception Dispatching_Policy_Error and none of the child packages that could raise that exception are pure. 17.c/4

Inconsistencies With Ada 2012

{*AI12-0299-1*} **Correction:** Substantially reduced the Implementation Permission that used to allow “altering” task dispatching points to only allow adding task dispatching points. If an implementation was using this permission to remove task dispatching points, and a program depended on that behavior to work, it could fail when used with Ada 2022. We are not aware of any such implementations, and such behavior was never portable to other implementations, so we do not expect this to matter in practice. 17.d/5

Extensions to Ada 2012

{*AI12-0279-1*} {*AI12-0294-1*} Aspect Yield is new. 17.e/5

Wording Changes from Ada 2012

{*AI12-0119-1*} Redid the description of task dispatching to include the separate threads of control that can be started by a parallel construct. 17.f/5

D.2.2 Task Dispatching Pragmas

{*AI95-00355-01*} {*AI05-0299-1*} [This subclause allows a single task dispatching policy to be defined for all priorities, or the range of priorities to be split into subranges that are assigned individual dispatching policies.] 0.1/3

Syntax

The form of a pragma Task_Dispatching_Policy is as follows: 1

pragma Task_Dispatching_Policy(*policy*_identifier); 2

{*AI95-00355-01*} The form of a pragma Priority_Specific_Dispatching is as follows: 2.1/2

pragma Priority_Specific_Dispatching (2.2/2
*policy*_identifier, *first_priority*_expression, *last_priority*_expression);

Name Resolution Rules

2.3/2 {*AI95-00355-01*} The expected type for *first_priority_*expression and *last_priority_*expression is Integer.

Legality Rules

3/2 {*AI95-00321-01*} {*AI95-00355-01*} The *policy_*identifier used in a pragma Task_Dispatching_Policy shall be the name of a task dispatching policy.

3.a/2 *This paragraph was deleted.*

3.1/2 {*AI95-00355-01*} The *policy_*identifier used in a pragma Priority_Specific_Dispatching shall be the name of a task dispatching policy.

3.2/2 {*AI95-00355-01*} Both *first_priority_*expression and *last_priority_*expression shall be static expressions in the range of System.Any_Priority; *last_priority_*expression shall have a value greater than or equal to *first_priority_*expression.

Static Semantics

3.3/2 {*AI95-00355-01*} Pragma Task_Dispatching_Policy specifies the single task dispatching policy.

3.4/2 {*AI95-00355-01*} Pragma Priority_Specific_Dispatching specifies the task dispatching policy for the specified range of priorities. Tasks with base priorities within the range of priorities specified in a Priority_Specific_Dispatching pragma have their active priorities determined according to the specified dispatching policy. Tasks with active priorities within the range of priorities specified in a Priority_Specific_Dispatching pragma are dispatched according to the specified dispatching policy.

3.b/2 **Reason:** {*AI95-00355-01*} Each ready queue is managed by exactly one policy. Anything else would be chaos. The ready queue is determined by the active priority. However, how the active priority is calculated is determined by the policy; in order to break out of this circle, we have to say that the active priority is calculated by the method determined by the policy of the base priority.

3.5/3 {*AI95-00355-01*} {*AI05-0262-1*} If a partition contains one or more Priority_Specific_Dispatching pragmas, the dispatching policy for priorities not covered by any Priority_Specific_Dispatching pragmas is FIFO_Within_Priorities.

Post-Compilation Rules

4/2 {*AI95-00355-01*} A Task_Dispatching_Policy pragma is a configuration pragma. A Priority_Specific_Dispatching pragma is a configuration pragma.

4.1/2 {*AI95-00355-01*} The priority ranges specified in more than one Priority_Specific_Dispatching pragma within the same partition shall not be overlapping.

4.2/2 {*AI95-00355-01*} If a partition contains one or more Priority_Specific_Dispatching pragmas it shall not contain a Task_Dispatching_Policy pragma.

5/2 *This paragraph was deleted.*{*AI95-00333-01*}

Dynamic Semantics

6/2 {*AI95-00355-01*} [A *task dispatching policy* specifies the details of task dispatching that are not covered by the basic task dispatching model. These rules govern when tasks are inserted into and deleted from the ready queues.] A single task dispatching policy is specified by a Task_Dispatching_Policy pragma. Pragma Priority_Specific_Dispatching assigns distinct dispatching policies to subranges of System.Any_Priority.

{*AI95-00355-01*} If neither pragma applies to any of the program units comprising a partition, the task dispatching policy for that partition is unspecified. 6.1/2

{*AI95-00355-01*} {*AI05-0262-1*} If a partition contains one or more Priority_Specific_Dispatching pragmas, a task dispatching point occurs for the currently running task of a processor whenever there is a nonempty ready queue for that processor with a higher priority than the priority of the running task. 6.2/3

Discussion: {*AI05-0005-1*} If we have priority specific dispatching then we want preemption across the entire range of priorities. That prevents higher priority tasks from being blocked by lower priority tasks that have a different policy. On the other hand, if we have a single policy for the entire partition, we want the characteristics of that policy to apply for preemption; specifically, we might not require any preemption. Note that policy Non_Preemptive_FIFO_Within_Priorities is not allowed in a priority specific dispatching pragma. 6.a/3

{*AI95-00355-01*} A task that has its base priority changed may move from one dispatching policy to another. It is immediately subject to the new dispatching policy. 6.3/2

Ramification: Once subject to the new dispatching policy, it may be immediately preempted or dispatched, according the rules of the new policy. 6.b/2

Paragraphs 7 through 13 were moved to D.2.3.

Implementation Requirements

{*AI95-00333-01*} {*AI95-00355-01*} An implementation shall allow, for a single partition, both the locking policy (see D.3) to be specified as Ceiling_Locking and also one or more Priority_Specific_Dispatching pragmas to be given. 13.1/2

Documentation Requirements

Paragraphs 14 through 16 were moved to D.2.3.

This paragraph was deleted. 16.a/2

Implementation Permissions

{*AI95-00256-01*} {*AI12-0444-1*} Implementations are allowed to define other task dispatching policies, but are not required to support specifying more than one task dispatching policy per partition. 17/5

{*AI95-00355-01*} {*AI12-0444-1*} An implementation is not required to support pragma Priority_Specific_Dispatching if it is infeasible to support it in the target environment. 18/5

Implementation defined: Implementation defined task dispatching policies. 18.a/2

Paragraphs 19 through 21 were deleted.

Extensions to Ada 95

{*AI95-00333-01*} **Amendment Correction:** It is no longer required to specify Ceiling_Locking with the language-defined task dispatching policies; we only require that implementations *allow* them to be used together. 21.a/2

{*AI95-00355-01*} {*AI05-0005-1*} **Pragma** Priority_Specific_Dispatching is new; it allows the specification of different policies for different priorities. 21.b/3

Wording Changes from Ada 95

{*AI95-00256-01*} Clarified that an implementation need support only one task dispatching policy (of any kind, language-defined or otherwise) per partition. 21.c/2

{*AI95-00321-01*} {*AI05-0005-1*} This description is simplified to describe only the rules for the Task_Dispatching_Policy pragma that are common to all policies. In particular, rules about preemption are moved elsewhere. This makes it easier to add other policies (which might not involve preemption). 21.d/3

D.2.3 Preemptive Dispatching

{*AI95-00321-01*} {*AI05-0299-1*} [This subclause defines a preemptive task dispatching policy.] 1/3

Static Semantics

2/2 {*AI95-00355-01*} The *policy_*identifier FIFO_Within_Priorities is a task dispatching policy.

Dynamic Semantics

3/2 {*AI95-00321-01*} When FIFO_Within_Priorities is in effect, modifications to the ready queues occur only as follows:

4/2 • {*AI95-00321-01*} When a blocked task becomes ready, it is added at the tail of the ready queue for its active priority.

5/2 • When the active priority of a ready task that is not running changes, or the setting of its base priority takes effect, the task is removed from the ready queue for its old active priority and is added at the tail of the ready queue for its new active priority, except in the case where the active priority is lowered due to the loss of inherited priority, in which case the task is added at the head of the ready queue for its new active priority.

6/2 • When the setting of the base priority of a running task takes effect, the task is added to the tail of the ready queue for its active priority.

7/2 • When a task executes a delay_statement that does not result in blocking, it is added to the tail of the ready queue for its active priority.

7.a/2 **Ramification:** If the delay does result in blocking, the task moves to the "delay queue", not to the ready queue.

8/2 {*AI95-00321-01*} Each of the events specified above is a task dispatching point (see D.2.1).

9/2 {*AI95-00321-01*} A task dispatching point occurs for the currently running task of a processor whenever there is a nonempty ready queue for that processor with a higher priority than the priority of the running task. The currently running task is said to be *preempted* and it is added at the head of the ready queue for its active priority.

Implementation Requirements

10/2 {*AI95-00333-01*} An implementation shall allow, for a single partition, both the task dispatching policy to be specified as FIFO_Within_Priorities and also the locking policy (see D.3) to be specified as Ceiling_Locking.

10.a/2 **Reason:** This is the preferred combination of the FIFO_Within_Priorities policy with a locking policy, and we want that combination to be portable.

Documentation Requirements

11/2 {*AI95-00321-01*} *Priority inversion* is the duration for which a task remains at the head of the highest priority nonempty ready queue while the processor executes a lower priority task. The implementation shall document:

12/2 • The maximum priority inversion a user task can experience due to activity of the implementation (on behalf of lower priority tasks), and

12.a/2 **Documentation Requirement:** The maximum priority inversion a user task can experience from the implementation.

13/2 • whether execution of a task can be preempted by the implementation processing of delay expirations for lower priority tasks, and if so, for how long.

13.a/2 **Documentation Requirement:** The amount of time that a task can be preempted for processing on behalf of lower-priority tasks.

14/2 NOTE 1 {*AI95-00321-01*} If the active priority of a running task is lowered due to loss of inherited priority (as it is on completion of a protected operation) and there is a ready task of the same active priority that is not running, the running task continues to run (provided that there is no higher priority task).

15/2 NOTE 2 {*AI95-00321-01*} Setting the base priority of a ready task causes the task to move to the tail of the queue for its active priority, regardless of whether the active priority of the task actually changes.

Wording Changes from Ada 95

{*AI95-00321-01*} This subclause is new; it mainly consists of text that was found in D.2.1 and D.2.2 in Ada 95. This was separated out so the definition of additional policies was easier. 15.a/2

{*AI95-00333-01*} We require that implementations allow this policy and Ceiling_Locking together. 15.b/2

{*AI95-00355-01*} We explicitly defined FIFO_Within_Priorities to be a task dispatching policy. 15.c/2

D.2.4 Non-Preemptive Dispatching

{*AI95-00298-01*} {*AI05-0299-1*} [This subclause defines a non-preemptive task dispatching policy.] 1/3

Static Semantics

{*AI95-00298-01*} {*AI95-00355-01*} The *policy*_identifier Non_Preemptive_FIFO_Within_Priorities is a task dispatching policy. 2/2

{*AI05-0166-1*} The following language-defined library package exists: 2.1/3

```
{AI12-0241-1} {AI12-0302-1} package Ada.Dispatching.Non_Preemptive                2.2/5
  with Preelaborate, Nonblocking, Global => in out synchronized is
  procedure Yield_To_Higher;
  procedure Yield_To_Same_Or_Higher renames Yield;
end Ada.Dispatching.Non_Preemptive;
```

{*AI05-0166-1*} {*AI05-0264-1*} A call of Yield_To_Higher is a task dispatching point for this policy. If the task at the head of the highest priority ready queue has a higher active priority than the calling task, then the calling task is preempted. 2.3/3

Ramification: For language-defined policies other than Non_Preemptive_FIFO_Within_Priorities, a higher priority task should never be on a ready queue while a lower priority task is executed. Thus, for such policies, Yield_To_Higher does nothing. 2.a/3

Yield_To_Higher is *not* a potentially blocking operation; it can be used during a protected operation. That is allowed, as under the predefined Ceiling_Locking policy any task with a higher priority than the protected operation cannot call the operation (that would violate the locking policy). An implementation-defined locking policy may need to define the semantics of Yield_To_Higher differently. 2.b/3

Legality Rules

{*AI95-00355-01*} Non_Preemptive_FIFO_Within_Priorities shall not be specified as the *policy*_identifier of pragma Priority_Specific_Dispatching (see D.2.2). 3/2

Reason: The non-preemptive nature of this policy could cause the policies of higher priority tasks to malfunction, missing deadlines and having unlimited priority inversion. That would render the use of such policies impotent and misleading. As such, this policy only makes sense for a complete system. 3.a/2

Dynamic Semantics

{*AI95-00298-01*} When Non_Preemptive_FIFO_Within_Priorities is in effect, modifications to the ready queues occur only as follows: 4/2

- {*AI95-00298-01*} When a blocked task becomes ready, it is added at the tail of the ready queue for its active priority. 5/2
- When the active priority of a ready task that is not running changes, or the setting of its base priority takes effect, the task is removed from the ready queue for its old active priority and is added at the tail of the ready queue for its new active priority. 6/2
- When the setting of the base priority of a running task takes effect, the task is added to the tail of the ready queue for its active priority. 7/2
- When a task executes a delay_statement that does not result in blocking, it is added to the tail of the ready queue for its active priority. 8/2

8.a/2 **Ramification:** If the delay does result in blocking, the task moves to the “delay queue”, not to the ready queue.

9/3 {*AI05-0166-1*} For this policy, blocking or termination of a task, a delay_statement, a call to Yield_To_Higher, and a call to Yield_To_Same_Or_Higher or Yield are the only task dispatching points (see D.2.1).

9.a/3 **Ramification:** {*AI05-0166-1*} A delay_statement is always a task dispatching point even if it is not blocking. Similarly, a call to Yield_To_Higher is never blocking, but it is a task dispatching point In each of these cases, they can cause the current task to stop running (it is still ready). Otherwise, the running task continues to run until it is blocked.

9.b/5 {*AI12-0299-1*} This rule supersedes the Implementation Permission of D.2.1; an implementation that adds additional task dispatching points to this policy is incorrect.

Implementation Requirements

10/2 {*AI95-00333-01*} An implementation shall allow, for a single partition, both the task dispatching policy to be specified as Non_Preemptive_FIFO_Within_Priorities and also the locking policy (see D.3) to be specified as Ceiling_Locking.

10.a/2 **Reason:** This is the preferred combination of the Non_Preemptive_FIFO_Within_Priorities policy with a locking policy, and we want that combination to be portable.

Implementation Permissions

11/3 {*AI95-00298-01*} {*AI05-0229-1*} {*AI05-0269-1*} Since implementations are allowed to round all ceiling priorities in subrange System.Priority to System.Priority'Last (see D.3), an implementation may allow a task of a partition using the Non_Premptive_FIFO_Within_Priorities policy to execute within a protected object without raising its active priority provided the associated protected unit does not contain any subprograms with aspects Interrupt_Handler or Attach_Handler specified, nor does the unit have aspect Interrupt_Priority specified. When the locking policy (see D.3) is Ceiling_Locking, an implementation taking advantage of this permission shall ensure that a call to Yield_to_Higher that occurs within a protected action uses the ceiling priority of the protected object (rather than the active priority of the task) when determining whether to preempt the task.

11.a.1/3 **Reason:** {*AI05-0269-1*} We explicitly require that the ceiling priority be used in calls to Yield_to_Higher in order to prevent a risk of priority inversion and consequent loss of mutual exclusion when Yield_to_Higher is used in a protected object. This requirement might lessen the value of the permission (as the current Ceiling_Priority will have to be maintained in the TCB), but loss of mutual exclusion cannot be tolerated. The primary benefit of the permission (eliminating the need for preemption at the end of a protected action) is still available. As noted above, an implementation-defined locking policy will need to specify the semantics of Yield_to_Higher, including this case.

Extensions to Ada 95

11.a/2 {*AI95-00298-01*} {*AI95-00355-01*} Policy Non_Preemptive_FIFO_Within_Priorities is new.

Extensions to Ada 2005

11.b/3 {*AI05-0166-1*} Package Dispatching.Non_Preemptive is new.

D.2.5 Round Robin Dispatching

1/3 {*AI95-00355-01*} {*AI05-0299-1*} [This subclause defines the task dispatching policy Round_Robin_Within_Priorities and the package Round_Robin.]

Static Semantics

2/2 {*AI95-00355-01*} The *policy*_identifier Round_Robin_Within_Priorities is a task dispatching policy.

{*AI95-00355-01*} The following language-defined library package exists: 3/2

```
{AI12-0241-1} {AI12-0302-1} with System;                                    4/5
with Ada.Real_Time;
package Ada.Dispatching.Round_Robin
  with Nonblocking, Global => in out synchronized is
  Default_Quantum : constant Ada.Real_Time.Time_Span :=
            implementation-defined;
  procedure Set_Quantum (Pri     : in System.Priority;
                         Quantum : in Ada.Real_Time.Time_Span);
  procedure Set_Quantum (Low, High : in System.Priority;
                         Quantum   : in Ada.Real_Time.Time_Span);
  function Actual_Quantum (Pri : System.Priority)
            return Ada.Real_Time.Time_Span;
  function Is_Round_Robin (Pri : System.Priority) return Boolean;
end Ada.Dispatching.Round_Robin;
```

Implementation defined: The value of Default_Quantum in Dispatching.Round_Robin. 4.a.1/2

{*AI95-00355-01*} When task dispatching policy Round_Robin_Within_Priorities is the single policy in effect for a partition, each task with priority in the range of System.Interrupt_Priority is dispatched according to policy FIFO_Within_Priorities. 5/2

Dynamic Semantics

{*AI95-00355-01*} The procedures Set_Quantum set the required Quantum value for a single priority level Pri or a range of priority levels Low .. High. If no quantum is set for a Round Robin priority level, Default_Quantum is used. 6/2

{*AI95-00355-01*} The function Actual_Quantum returns the actual quantum used by the implementation for the priority level Pri. 7/2

{*AI95-00355-01*} {*AI05-0264-1*} The function Is_Round_Robin returns True if priority Pri is covered by task dispatching policy Round_Robin_Within_Priorities; otherwise, it returns False. 8/3

{*AI95-00355-01*} A call of Actual_Quantum or Set_Quantum raises exception Dispatching.Dispatching_Policy_Error if a predefined policy other than Round_Robin_Within_Priorities applies to the specified priority or any of the priorities in the specified range. 9/2

{*AI95-00355-01*} For Round_Robin_Within_Priorities, the dispatching rules for FIFO_Within_Priorities apply with the following additional rules: 10/2

- When a task is added or moved to the tail of the ready queue for its base priority, it has an execution time budget equal to the quantum for that priority level. This will also occur when a blocked task becomes executable again. 11/2
- When a task is preempted (by a higher priority task) and is added to the head of the ready queue for its priority level, it retains its remaining budget. 12/2
- While a task is executing, its budget is decreased by the amount of execution time it uses. The accuracy of this accounting is the same as that for execution time clocks (see D.14). 13/2

  Ramification: Note that this happens even when the task is executing at a higher, inherited priority, and even if that higher priority is dispatched by a different policy than round robin. 13.a/2
- When a task has exhausted its budget and is without an inherited priority (and is not executing within a protected operation), it is moved to the tail of the ready queue for its priority level. This is a task dispatching point. 14/2

  Ramification: In this case, it will be given a budget as described in the first bullet. 14.a/2

  The rules for FIFO_Within_Priority (to which these bullets are added) say that a task that has its base priority set to a Round Robin priority is moved to the tail of the ready queue for its new priority level, and then will be given a budget as described in the first bullet. That happens whether or not the task's original base priority was a Round Robin priority. 14.b/2

Implementation Requirements

15/2 {*AI95-00333-01*} {*AI95-00355-01*} An implementation shall allow, for a single partition, both the task dispatching policy to be specified as Round_Robin_Within_Priorities and also the locking policy (see D.3) to be specified as Ceiling_Locking.

15.a/2 **Reason:** This is the preferred combination of the Round_Robin_Within_Priorities policy with a locking policy, and we want that combination to be portable.

Documentation Requirements

16/2 {*AI95-00355-01*} An implementation shall document the quantum values supported.

16.a.1/2 **Documentation Requirement:** The quantum values supported for round robin dispatching.

17/2 {*AI95-00355-01*} An implementation shall document the accuracy with which it detects the exhaustion of the budget of a task.

17.a.1/2 **Documentation Requirement:** The accuracy of the detection of the exhaustion of the budget of a task for round robin dispatching.

18/5 NOTE 1 {*AI95-00355-01*} {*AI12-0442-1*} Due to implementation constraints, the quantum value returned by Actual_Quantum can differ from that set with Set_Quantum.

19/2 NOTE 2 {*AI95-00355-01*} A task that executes continuously with an inherited priority will not be subject to round robin dispatching.

Extensions to Ada 95

19.a/2 {*AI95-00355-01*} Policy Round_Robin_Within_Priorities and package Dispatching.Round_Robin are new.

D.2.6 Earliest Deadline First Dispatching

1/5 {*AI95-00357-01*} {*AI12-0439-1*} The deadline of a task is an indication of the urgency of the task; it represents a point on an ideal physical time line. The deadline can affect how resources are allocated to the task.

2/5 {*AI95-00357-01*} {*AI05-0229-1*} {*AI05-0299-1*} {*AI12-0230-1*} [This subclause presents Dispatching.EDF, a package for representing the deadline of a task and a dispatching policy that defines Earliest Deadline First (EDF) dispatching. The Relative_Deadline aspect is provided to assign an initial deadline to a task. A configuration pragma Generate_Deadlines is provided to specify that a task's deadline is recomputed whenever it is made ready.]

2.a/5 *This paragraph was deleted.*{*AI05-0229-1*} {*AI12-0230-1*}

Language Design Principles

2.b/5 *This paragraph was deleted.*{*AI95-00357-01*} {*AI05-0299-1*} {*AI12-0230-1*}

Paragraphs 3 through 6 were moved to Annex J, "Obsolescent Features".

Static Semantics

7/5 {*AI95-00357-01*} {*AI12-0230-1*} The *policy_*identifier EDF_Within_Priorities is a task dispatching policy.

{*AI95-00357-01*} The following language-defined library package exists: 8/2

```
{AI12-0230-1} {AI12-0241-1} {AI12-0302-1} with Ada.Real_Time;                    9/5
with Ada.Task_Identification;
package Ada.Dispatching.EDF
  with Nonblocking, Global => in out synchronized is
  subtype Deadline is Ada.Real_Time.Time;
  subtype Relative_Deadline is Ada.Real_Time.Time_Span;
  Default_Deadline : constant Deadline :=
              Ada.Real_Time.Time_Last;
  Default_Relative_Deadline : constant Relative_Deadline :=
              Ada.Real_Time.Time_Span_Last;
  procedure Set_Deadline
     (D : in Deadline;
      T : in Ada.Task_Identification.Task_Id :=
      Ada.Task_Identification.Current_Task);
  function Get_Deadline
     (T : Ada.Task_Identification.Task_Id :=
      Ada.Task_Identification.Current_Task) return Deadline;
  procedure Set_Relative_Deadline
     (D : in Relative_Deadline;
      T : in Ada.Task_Identification.Task_Id :=
      Ada.Task_Identification.Current_Task);
  function Get_Relative_Deadline
     (T : Ada.Task_Identification.Task_Id :=
      Ada.Task_Identification.Current_Task)
      return Relative_Deadline;
  procedure Delay_Until_And_Set_Deadline
     (Delay_Until_Time : in Ada.Real_Time.Time;
      Deadline_Offset : in Ada.Real_Time.Time_Span)
     with Nonblocking => False;
  function Get_Last_Release_Time
     (T : Ada.Task_Identification.Task_Id :=
      Ada.Task_Identification.Current_Task)
      return Ada.Real_Time.Time;
end Ada.Dispatching.EDF;
```

{*AI05-0229-1*} {*AI12-0230-1*} For a subprogram, a task type (including the anonymous type of a single_task_declaration), or a protected type (including the anonymous type of a single_protected_declaration), the following language-defined representation aspect may be specified: 9.1/5

Relative_Deadline 9.2/3

The aspect Relative_Deadline is an expression, which shall be of type Real_Time.Time_Span.

Aspect Description for Relative_Deadline: Task or protected type parameter used in Earliest Deadline First Dispatching. 9.a/5

{*AI12-0230-1*} The form of pragma Generate_Deadlines is as follows: 9.3/5

pragma Generate_Deadlines; 9.4/5

{*AI12-0230-1*} The Generate_Deadlines pragma is a configuration pragma. 9.5/5

Legality Rules

{*AI05-0229-1*} {*AI12-0230-1*} The Relative_Deadline aspect shall not be specified on a task or protected interface type. If the Relative_Deadline aspect is specified for a subprogram, the aspect_definition shall be a static expression. 9.6/5

Post-Compilation Rules

{*AI95-00357-01*} {*AI12-0230-1*} If the EDF_Within_Priorities policy is specified for a partition, then the Ceiling_Locking policy (see D.3) shall also be specified for the partition. 10/5

11/5 {*AI95-00357-01*} {*AI12-0230-1*} If the EDF_Within_Priorities policy appears in a Priority_Specific_Dispatching pragma (see D.2.2) in a partition, then the Ceiling_Locking policy (see D.3) shall also be specified for the partition.

11.a/5 **Reason:** Unlike the other language-defined dispatching policies, the semantic description of EDF_Within_Priorities assumes Ceiling_Locking (and a ceiling priority) in order to make the mapping between deadlines and priorities work. Thus, we require both policies to be specified if EDF is used in the partition.

Dynamic Semantics

12/3 {*AI95-00357-01*} {*AI05-0229-1*} The Relative_Deadline aspect has no effect if it is specified for a subprogram other than the main subprogram.

12.1/5 {*AI12-0230-1*} If pragma Generate_Deadlines is in effect, the deadline of a task is recomputed each time it becomes ready. The new deadline is the value of Real_Time.Clock at the time the task is added to a ready queue plus the value returned by Get_Relative_Deadline.

13/5 {*AI95-00357-01*} {*AI05-0229-1*} {*AI12-0230-1*} The initial absolute deadline for a task with a specified Relative_Deadline is the result of adding the value returned by a call of Real_Time.Clock to the value of the expression specified as the Relative_Deadline aspect, where this entire computation, including the call of Real_Time.Clock, is performed between task creation and the start of its activation. If the aspect Relative_Deadline is not specified, then the initial absolute deadline of a task is the value of Default_Deadline (Ada.Real_Time.Time_Last). The environment task is also given an initial deadline by this rule, using the value of the Relative_Deadline aspect of the main subprogram (if any).

13.a/2 **Proof:** The environment task is a normal task by 10.2, so of course this rule applies to it.

13.1/5 {*AI12-0230-1*} The effect of specifying a Relative_Deadline aspect for a protected type or single_protected_declaration is discussed in D.3.

14/5 {*AI95-00357-01*} {*AI12-0230-1*} A task has both an *active* and a *base* absolute deadline. These are the same except when the task is inheriting a relative deadline during activation or a rendezvous (see below) or within a protected action (see D.3). The procedure Set_Deadline changes the (base) absolute deadline of the task to D. The function Get_Deadline returns the (base) absolute deadline of the task.

14.1/5 {*AI12-0230-1*} The procedure Set_Relative_Deadline changes the relative deadline of the task to D. The function Get_Relative_Deadline returns the relative deadline of the task.

14.2/5 {*AI12-0230-1*} The function Get_Last_Release_Time returns the time, as provided by Real_Time.Clock, when the task was last made ready (that is, was added to a ready queue).

15/5 {*AI95-00357-01*} {*AI12-0230-1*} The procedure Delay_Until_And_Set_Deadline delays the calling task until time Delay_Until_Time. When the task becomes ready again it will have deadline `Delay_Until_Time + Deadline_Offset`.

16/5 {*AI95-00357-01*} {*AI12-0230-1*} On a system with a single processor, the setting of the deadline of a task to the new value occurs immediately at the first point that is outside the execution of a protected action. If the task is currently on a ready queue it is removed and re-entered onto the ready queue determined by the rules defined below.

17/5 {*AI95-00357-01*} {*AI12-0230-1*} When EDF_Within_Priorities is specified for a priority, the ready queue for that priority is ordered by deadline. The task at the head of a queue is the one with the earliest deadline.

18/5 {*AI95-00357-01*} {*AI12-0230-1*} A task dispatching point occurs for the currently running task *T* to which policy EDF_Within_Priorities applies:

19/5 - {*AI12-0230-1*} when a change to the base (absolute) deadline of *T* occurs;

- *This paragraph was deleted.*{*AI12-0230-1*} 20/5
- {*AI12-0230-1*} there is a nonempty ready queue for that processor with a higher priority than the active priority of the running task; 21/5
- {*AI12-0230-1*} there is a ready task with the same priority as *T* but with an earlier absolute deadline. 21.1/5

{*AI12-0230-1*} In these cases, the currently running task is said to be preempted and is returned to the ready queue for its active priority, at a position determined by its active (absolute) deadline. 22/5

Paragraphs 23 through 27 were deleted.

{*AI95-00357-01*} {*AI12-0230-1*} When the setting of the base priority of a ready task takes effect and the new priority is specified as EDF_Within_Priorities, the task is added to the ready queue, at a position determined by its active deadline. 28/5

{*AI95-00357-01*} For all the operations defined in Dispatching.EDF, Tasking_Error is raised if the task identified by T has terminated. Program_Error is raised if the value of T is Null_Task_Id. 29/2

{*AI12-0230-1*} If two tasks with priority designated as EDF_Within_Priorities rendezvous then the deadline for the execution of the accept statement is the earlier of the deadlines of the two tasks. 29.1/5

{*AI12-0230-1*} During activation, a task being activated inherits the deadline that its activator (see 9.2) had at the time the activation was initiated. 29.2/5

Paragraph 30 was deleted.

Erroneous Execution

{*AI95-00357-01*} If a value of Task_Id is passed as a parameter to any of the subprograms of this package and the corresponding task object no longer exists, the execution of the program is erroneous. 31/2

Documentation Requirements

{*AI95-00357-01*} On a multiprocessor, the implementation shall document any conditions that cause the completion of the setting of the deadline of a task to be delayed later than what is specified for a single processor. 32/2

Documentation Requirement: Any conditions that cause the completion of the setting of the deadline of a task to be delayed for a multiprocessor. 32.a.1/2

NOTE {*AI95-00357-01*} {*AI05-0264-1*} {*AI12-0230-1*} If two distinct priorities are specified to have policy EDF_Within_Priorities, then tasks from the higher priority always run before tasks of the lower priority, regardless of deadlines. 33/5

This paragraph was deleted.{*AI95-00357-01*} {*AI12-0230-1*} 34/5

Implementation Note: {*AI95-00357-01*} An implementation may support additional dispatching policies by replacing absolute deadline with an alternative measure of urgency. 34.a/2

Extensions to Ada 95

{*AI95-00357-01*} Policy EDF_Across_Priorities and package Dispatching.EDF are new. 34.b/2

Extensions to Ada 2005

{*AI05-0229-1*} Aspect Relative_Deadline is new; pragma Relative_Deadline is now obsolescent. 34.c/3

Wording Changes from Ada 2005

{*AI05-0055-1*} **Correction:** Corrected definition of active priority to avoid deadline inversion in an unusual case. 34.d/3

Incompatibilities With Ada 2012

34.e/5 {*AI12-0005-1*} {*AI12-0230-1*} The policy EDF_Across_Priorities was replaced by EDF_Within_Priorities. A program using EDF_Across_Priorities could fail to compile. However, we are not aware of any implementations of EDF_Across_Priorities, so it seems unlikely that any such programs exist outside of books and papers.

D.3 Priority Ceiling Locking

1/3 {*AI05-0299-1*} [This subclause specifies the interactions between priority task scheduling and protected object ceilings. This interaction is based on the concept of the *ceiling priority* of a protected object.]

Syntax

2 The form of a pragma Locking_Policy is as follows:

3 **pragma** Locking_Policy(*policy*_identifier);

Legality Rules

4 The *policy*_identifier shall either be Ceiling_Locking or an implementation-defined identifier.

4.a **Implementation defined:** Implementation-defined *policy*_identifiers allowed in a pragma Locking_Policy.

Post-Compilation Rules

5 A Locking_Policy pragma is a configuration pragma.

Dynamic Semantics

6/2 {*8652/0073*} {*AI95-00091-01*} {*AI95-00327-01*} [A locking policy specifies the details of protected object locking. All protected objects have a priority. The locking policy specifies the meaning of the priority of a protected object, and the relationships between these priorities and task priorities. In addition, the policy specifies the state of a task when it executes a protected action, and how its active priority is affected by the locking.] The *locking policy* is specified by a Locking_Policy pragma. For implementation-defined locking policies, the meaning of the priority of a protected object is implementation defined. If no Locking_Policy pragma applies to any of the program units comprising a partition, the locking policy for that partition, as well as the meaning of the priority of a protected object, are implementation defined.

6.a/2 **Implementation defined:** The locking policy if no Locking_Policy pragma applies to any unit of a partition.

6.1/3 {*AI95-00327-01*} {*AI05-0229-1*} The expression specified for the Priority or Interrupt_Priority aspect (see D.1) is evaluated as part of the creation of the corresponding protected object and converted to the subtype System.Any_Priority or System.Interrupt_Priority, respectively. The value of the expression is the initial priority of the corresponding protected object. If no Priority or Interrupt_Priority aspect is specified for a protected object, the initial priority is specified by the locking policy.

7 There is one predefined locking policy, Ceiling_Locking; this policy is defined as follows:

8/3 - {*AI95-00327-01*} {*AI05-0229-1*} Every protected object has a *ceiling priority*, which is determined by either a Priority or Interrupt_Priority aspect as defined in D.1, or by assignment to the Priority attribute as described in D.5.2. The ceiling priority of a protected object (or ceiling, for short) is an upper bound on the active priority a task can have when it calls protected operations of that protected object.

9/2 - {*AI95-00327-01*} The initial ceiling priority of a protected object is equal to the initial priority for that object.

10/4 - {*AI95-00327-01*} {*AI05-0229-1*} {*AI12-0051-1*} If an Interrupt_Handler or Attach_Handler aspect (see C.3.1) is specified for a protected subprogram of a protected type that does not have

either the Priority or Interrupt_Priority aspect specified, the initial priority of protected objects of that type is implementation defined, but in the range of the subtype System.Interrupt_Priority.

Implementation defined: Default ceiling priorities. 10.a

- {*AI95-00327-01*} {*AI05-0229-1*} If neither aspect Priority nor Interrupt_Priority is specified for a protected type, and no protected subprogram of the type has aspect Interrupt_Handler or Attach_Handler specified, then the initial priority of the corresponding protected object is System.Priority'Last. 11/3
- While a task executes a protected action, it inherits the ceiling priority of the corresponding protected object. 12
- When a task calls a protected operation, a check is made that its active priority is not higher than the ceiling of the corresponding protected object; Program_Error is raised if this check fails. 13

{*AI12-0230-1*} {*AI12-0404-1*} If the task dispatching policy specified for the ceiling priority of a protected object is EDF_Within_Priorities, the following additional rules apply: 13.1/5

- Every protected object has a *relative deadline*, which is determined by a Relative_Deadline aspect as defined in D.2.6, or by assignment to the Relative_Deadline attribute as described in D.5.2. The relative deadline of a protected object represents a lower bound on the relative deadline a task may have when it calls a protected operation of that protected object. 13.2/5
- If aspect Relative_Deadline is not specified for a protected type then the initial relative deadline of the corresponding protected object is Ada.Real_Time.Time_Span_Zero. 13.3/5
- While a task executes a protected action on a protected object *P*, it inherits the relative deadline of *P*. In this case, let *DF* be 'now' ('now' is obtained via a call on Ada.Real_Time.Clock at the start of the action) plus the deadline floor of *P*. If the active deadline of the task is later than *DF*, its active deadline is reduced to *DF*[; the active deadline is unchanged otherwise]. 13.4/5
- When a task calls a protected operation, a check is made that its active deadline minus its last release time is not less than the relative deadline of the corresponding protected object; Program_Error is raised if this check fails. 13.5/5

Bounded (Run-Time) Errors

{*AI95-00327-01*} {*AI12-0230-1*} Following any change of priority, it is a bounded error for the active priority of any task with a call queued on an entry of a protected object to be higher than the ceiling priority of the protected object. In this case one of the following applies: 13.6/5

- {*AI12-0404-1*} at any time prior to executing the entry body, Program_Error is raised in the calling task; 13.7/5
- {*AI12-0404-1*} when the entry is open, the entry body is executed at the ceiling priority of the protected object; 13.8/5
- {*AI12-0404-1*} when the entry is open, the entry body is executed at the ceiling priority of the protected object and then Program_Error is raised in the calling task; or 13.9/5
- {*AI12-0404-1*} when the entry is open, the entry body is executed at the ceiling priority of the protected object that was in effect when the entry call was queued. 13.10/5

Ramification: Note that the error is “blamed” on the task that did the entry call, not the task that changed the priority of the task or protected object. This seems to make sense for the case of changing the priority of a task blocked on a call, since if the Set_Priority had happened a little bit sooner, before the task queued a call, the entry-calling task would get the error. Similarly, there is no reason not to raise the priority of a task that is executing in an abortable_part, so long as its priority is lowered before it gets to the end and needs to cancel the call. The priority might need to be lowered to allow it to remove the call from the entry queue, in order to avoid violating the ceiling. This seems relatively harmless, since there is an error, and the task is about to start raising an exception anyway. 13.a.1/2

Implementation Permissions

14 The implementation is allowed to round all ceilings in a certain subrange of System.Priority or System.Interrupt_Priority up to the top of that subrange, uniformly.

14.a **Discussion:** For example, an implementation might use Priority'Last for all ceilings in Priority, and Interrupt_Priority'Last for all ceilings in Interrupt_Priority. This would be equivalent to having two ceiling priorities for protected objects, “nonpreemptible” and “noninterruptible”, and is an allowed behavior.

14.b Note that the implementation cannot choose a subrange that crosses the boundary between normal and interrupt priorities.

15/5 {*AI95-00256-01*} {*AI12-0444-1*} Implementations are allowed to define other locking policies, but are not required to support specifying more than one locking policy per partition.

16 [Since implementations are allowed to place restrictions on code that runs at an interrupt-level active priority (see C.3.1 and D.2.1), the implementation may implement a language feature in terms of a protected object with an implementation-defined ceiling, but the ceiling shall be no less than Priority'Last.]

16.a **Implementation defined:** The ceiling of any protected object used internally by the implementation.

16.b **Proof:** This permission follows from the fact that the implementation can place restrictions on interrupt handlers and on any other code that runs at an interrupt-level active priority.

16.c The implementation might protect a storage pool with a protected object whose ceiling is Priority'Last, which would cause allocators to fail when evaluated at interrupt priority. Note that the ceiling of such an object has to be at least Priority'Last, since there is no permission for allocators to fail when evaluated at a noninterrupt priority.

Implementation Advice

17 The implementation should use names that end with “_Locking” for implementation-defined locking policies.

17.a/2 **Implementation Advice:** Names that end with “_Locking” should be used for implementation-defined locking policies.

18 NOTE 1 While a task executes in a protected action, it can be preempted only by tasks whose active priorities are higher than the ceiling priority of the protected object.

19 NOTE 2 If a protected object has a ceiling priority in the range of Interrupt_Priority, certain interrupts are blocked while protected actions of that object execute. In the extreme, if the ceiling is Interrupt_Priority'Last, all blockable interrupts are blocked during that time.

20/5 NOTE 3 {*AI12-0447-1*} As described in C.3.1, whenever an interrupt is handled by one of the protected procedures of a protected object, a check is made that its ceiling priority is in the Interrupt_Priority range.

21/5 NOTE 4 {*AI12-0442-1*} When specifying the ceiling of a protected object, a correct value is one that is at least as high as the highest active priority at which tasks can be executing when they call protected operations of that object. In determining this value the following factors, which can affect active priority, are relevant: the effect of Set_Priority, nested protected operations, entry calls, task activation, and other implementation-defined factors.

22 NOTE 5 Attaching a protected procedure whose ceiling is below the interrupt hardware priority to an interrupt causes the execution of the program to be erroneous (see C.3.1).

23 NOTE 6 On a single processor implementation, the ceiling priority rules guarantee that there is no possibility of deadlock involving only protected subprograms (excluding the case where a protected operation calls another protected operation on the same protected object).

Extensions to Ada 95

23.a/2 {*AI95-00327-01*} All protected objects now have a priority, which is the value of the Priority attribute of D.5.2. How this value is interpreted depends on the locking policy; for instance, the ceiling priority is derived from this value when the locking policy is Ceiling_Locking.

Wording Changes from Ada 95

23.b/2 {*8652/0073*} {*AI95-00091-01*} **Corrigendum:** Corrected the wording to reflect that pragma Locking_Policy cannot be inside of a program unit.

{*AI95-00256-01*} Clarified that an implementation need support only one locking policy (of any kind, language-defined or otherwise) per partition. 23.c/2

{*AI95-00327-01*} The bounded error for the priority of a task being higher than the ceiling of an object it is currently in was moved here from D.5, so that it applies no matter how the situation arises. 23.d/2

Wording Changes from Ada 2005

{*AI05-0229-1*} Revised to use aspects Priority and Interrupt_Priority as pragmas Priority and Interrupt_Priority are now obsolescent. 23.e/3

Extensions to Ada 2012

{*AI12-0230-1*} All protected objects now have a relative deadline, which is the value of the Relative_Deadline attribute of D.5.2. How this value is interpreted depends on the locking policy. 23.f/5

Wording Changes from Ada 2012

{*AI12-0051-1*} **Corrigendum:** Clarified that the Priority aspect can be used to set the initial ceiling priority of a protected object that contains an interrupt handler. 23.g/4

D.4 Entry Queuing Policies

{*8652/0074*} {*AI95-00068-01*} {*AI05-0299-1*} {*AI12-0324-1*} [This subclause specifies a mechanism for a user to choose an entry *queuing policy*. It also defines three such policies. Other policies are implementation defined.] 1/5

Implementation defined: Implementation-defined queuing policies. 1.a

Syntax

The form of a pragma Queuing_Policy is as follows: 2

pragma Queuing_Policy(*policy*_identifier); 3

Legality Rules

{*AI12-0183-1*} The *policy*_identifier shall be either FIFO_Queuing, Ordered_FIFO_Queuing, Priority_Queuing or an implementation-defined identifier. 4/5

Post-Compilation Rules

A Queuing_Policy pragma is a configuration pragma. 5

Dynamic Semantics

[A *queuing policy* governs the order in which tasks are queued for entry service, and the order in which different entry queues are considered for service.] The queuing policy is specified by a Queuing_Policy pragma. 6

Ramification: The queuing policy includes entry queuing order, the choice among open alternatives of a selective_accept, and the choice among queued entry calls of a protected object when more than one entry_barrier condition is True. 6.a

{*AI95-00355-01*} {*AI12-0163-1*} {*AI12-0183-1*} Three queuing policies, FIFO_Queuing, Ordered_FIFO_Queuing, and Priority_Queuing, are language defined. If no Queuing_Policy pragma applies to any of the program units comprising the partition, the queuing policy for that partition is FIFO_Queuing. The rules for the FIFO_Queuing policy are specified in 9.5.3 and 9.7.1. 7/5

{*AI12-0163-1*} The Ordered_FIFO_Queuing policy is defined as follows: 7.1/5

- Calls are selected on a given entry queue in order of arrival. 7.2/5

7.3/5 • When more than one condition of an entry_barrier of a protected object becomes True, and more than one of the respective queues is nonempty, the call that arrived first is selected.

7.4/5 • If the expiration time of two or more open delay_alternatives is the same and no other accept_alternatives are open, the sequence_of_statements of the delay_alternative that is first in textual order in the selective_accept is executed.

7.5/5 • When more than one alternative of a selective_accept is open and has queued calls, the alternative whose queue has the call that arrived first is selected.

7.a/5 **Implementation Note:** A possible implementation for this policy would be to assign a sequence number to each queued entry call, where the sequence number is incremented globally across all queues associated with the protected object or selective_accept.

7.b/5 **Reason:** It would have been marginally easier to use textual order for the case when multiple queues are selectable. But textual order can lead to unfair queue servicing, since the queues in earlier textual order will end up starving the later ones if calls arrive fast enough. Applying arrival first as the selector provides fairness for this policy.

8 The Priority_Queuing policy is defined as follows:

9 • The calls to an entry [(including a member of an entry family)] are queued in an order consistent with the priorities of the calls. The *priority of an entry call* is initialized from the active priority of the calling task at the time the call is made, but can change later. Within the same priority, the order is consistent with the calling (or requeuing, or priority setting) time (that is, a FIFO order).

10/1 • {*8652/0075*} {*AI95-00205-01*} After a call is first queued, changes to the active priority of a task do not affect the priority of the call, unless the base priority of the task is set while the task is blocked on an entry call.

11 • When the base priority of a task is set (see D.5), if the task is blocked on an entry call, and the call is queued, the priority of the call is updated to the new active priority of the calling task. This causes the call to be removed from and then reinserted in the queue at the new active priority.

11.a **Reason:** A task is blocked on an entry call if the entry call is simple, conditional, or timed. If the call came from the triggering_statement of an asynchronous_select, or a requeue thereof, then the task is not blocked on that call; such calls do not have their priority updated. Thus, there can exist many queued calls from a given task (caused by many nested ATC's), but a task can be blocked on only one call at a time.

11.b A previous version of Ada 9X required queue reordering in the asynchronous_select case as well. If the call corresponds to a “synchronous” entry call, then the task is blocked while queued, and it makes good sense to move it up in the queue if its priority is raised.

11.c However, if the entry call is “asynchronous”, that is, it is due to an asynchronous_select whose triggering_statement is an entry call, then the task is not waiting for this entry call, so the placement of the entry call on the queue is irrelevant to the rate at which the task proceeds.

11.d Furthermore, when an entry is used for asynchronous_selects, it is almost certain to be a “broadcast” entry or have only one caller at a time. For example, if the entry is used to notify tasks of a mode switch, then all tasks on the entry queue would be signaled when the mode changes. Similarly, if it is indicating some interrupting event such as a control-C, all tasks sensitive to the interrupt will want to be informed that the event occurred. Hence, the order on such a queue is essentially irrelevant.

11.e Given the above, it seems an unnecessary semantic and implementation complexity to specify that asynchronous queued calls are moved in response to dynamic priority changes. Furthermore, it is somewhat inconsistent, since the call was originally queued based on the active priority of the task, but dynamic priority changes are changing the base priority of the task, and only indirectly the active priority. We say explicitly that asynchronous queued calls are not affected by normal changes in active priority during the execution of an abortable_part. Saying that, if a change in the base priority affects the active priority, then we do want the calls reordered, would be inconsistent. It would also require the implementation to maintain a readily accessible list of all queued calls which would not otherwise be necessary.

11.f Several rules were removed or simplified when we changed the rules so that calls due to asynchronous_selects are never moved due to intervening changes in active priority, be they due to protected actions, some other priority inheritance, or changes in the base priority.

12 • When more than one condition of an entry_barrier of a protected object becomes True, and more than one of the respective queues is nonempty, the call with the highest priority is selected. If more than one such call has the same priority, the call that is queued on the entry whose

declaration is first in textual order in the protected_definition is selected. For members of the same entry family, the one with the lower family index is selected.

- If the expiration time of two or more open delay_alternatives is the same and no other accept_alternatives are open, the sequence_of_statements of the delay_alternative that is first in textual order in the selective_accept is executed. 13
- When more than one alternative of a selective_accept is open and has queued calls, an alternative whose queue has the highest-priority call at its head is selected. If two or more open alternatives have equal-priority queued calls, then a call on the entry in the accept_alternative that is first in textual order in the selective_accept is selected. 14

Implementation Permissions

{*AI95-00256-01*} {*AI12-0444-1*} Implementations are allowed to define other queuing policies, but are not required to support specifying more than one queuing policy per partition. 15/5

Discussion: {*8652/0116*} {*AI95-00069-01*} {*AI95-00256-01*} This rule is really redundant, as 10.1.5 allows an implementation to limit the use of configuration pragmas to an empty environment. In that case, there would be no way to have multiple policies in a partition. 15.a.1/2

{*AI95-00188-02*} Implementations are allowed to defer the reordering of entry queues following a change of base priority of a task blocked on the entry call if it is not practical to reorder the queue immediately. 15.1/2

Reason: Priority change is immediate, but the effect of the change on entry queues can be deferred. That is necessary in order to implement priority changes on top of a non-Ada kernel. 15.a.2/2

Discussion: The reordering should occur as soon as the blocked task can itself perform the reinsertion into the entry queue. 15.a.3/2

Implementation Advice

The implementation should use names that end with "_Queuing" for implementation-defined queuing policies. 16

Implementation Advice: Names that end with "_Queuing" should be used for implementation-defined queuing policies. 16.a/2

Static Semantics

{*AI12-0164-1*} For a task type (including the anonymous type of a single_task_declaration), protected type (including the anonymous type of a single_protected_declaration), or an entry_declaration, the following language-defined representation aspect may be specified: 17/5

Max_Entry_Queue_Length 18/5

The type of aspect Max_Entry_Queue_Length is Integer.

Aspect Description for Max_Entry_Queue_Length: The maximum entry queue length for a task type, protected type, or entry. 18.a/5

If directly specified, the aspect_definition shall be a static expression no less than -1. If not specified, the aspect has value -1 (representing no additional restriction on queue length). 19/5

Legality Rules

{*AI12-0164-1*} {*AI12-0388-1*} If the Max_Entry_Queue_Length aspect for a type has a nonnegative value, the Max_Entry_Queue_Length aspect for every individual entry of that type shall not be greater than the value of the aspect for the type. The Max_Entry_Queue_Length aspect of a type is nonoverridable (see 13.1.1). 20/5

Ramification: Aspect Max_Entry_Queue_Length can specify less than the partition-wide or type-wide default, but it can't expand the length of a queue. 20.a/5

Post-Compilation Rules

21/5 {*AI12-0164-1*} If a restriction Max_Entry_Queue_Length applies to a partition, any value specified for the Max_Entry_Queue_Length aspect specified for the declaration of a type or entry in the partition shall not be greater than the value of the restriction.

21.a/5 **Ramification:** 13.12(6) says that the restriction value has to be static, so this is statically checkable. But the restriction does not have to be in the same compilation as the aspect, so the check cannot, in general, be done until link time.

Dynamic Semantics

22/5 {*AI12-0164-1*} If a nonconfirming value is specified for Max_Entry_Queue_Length for a type, and an entry call or requeue would cause the queue for any entry of the type to become longer than the specified value, then Program_Error is raised at the point of the call or requeue.

23/5 {*AI12-0164-1*} If a nonconfirming value is specified for Max_Entry_Queue_Length for an entry, and an entry call or requeue would cause the queue for an entry to become longer than the specified value, then Program_Error is raised at the point of the call or requeue.

Wording Changes from Ada 95

23.a/2 {*8652/0074*} {*AI95-00068-01*} **Corrigendum:** Corrected the number of queuing policies defined.

23.b/2 {*8652/0075*} {*AI95-00205-01*} **Corrigendum:** Corrected so that a call of Set_Priority in an abortable part does not change the priority of the triggering entry call.

23.c/2 {*AI95-00188-02*} Added a permission to defer queue reordering when the base priority of a task is changed. This is a counterpart to stronger requirements on the implementation of priority change.

23.d/2 {*AI95-00256-01*} Clarified that an implementation need support only one queuing policy (of any kind, language-defined or otherwise) per partition.

23.e/2 {*AI95-00355-01*} Fixed wording to make clear that pragma never appears inside of a unit; rather it “applies to” the unit.

Extensions to Ada 2012

23.f/5 {*AI12-0163-1*} Defined the new queuing policy Ordered_FIFO_Queuing.

23.g/5 {*AI12-0164-1*} Defined the new aspect Max_Entry_Queue_Length.

D.4.1 Admission Policies

1/5 {*AI12-0276-1*} [This subclause specifies a mechanism for a user to choose an admission policy. It also defines one such policy. Other policies are implementation defined.]

1.a.1/5 **Implementation defined:** Implementation-defined admission policies.

Syntax

2/5 {*AI12-0276-1*} The form of a pragma Admission_Policy is as follows:

3/5 **pragma** Admission_Policy (*policy*_identifier);

Legality Rules

4/5 {*AI12-0276-1*} The *policy*_identifier shall be either FIFO_Spinning or an implementation-defined identifier.

Post-Compilation Rules

5/5 {*AI12-0276-1*} An Admission_Policy pragma is a configuration pragma.

Dynamic Semantics

{*AI12-0276-1*} An admission policy governs the order in which competing tasks are evaluated for acquiring the execution resource associated with a protected object. The admission policy is specified by an Admission_Policy pragma. 6/5

{*AI12-0276-1*} One admission policy, FIFO_Spinning, is language defined. If FIFO_Spinning is in effect, and starting a protected action on a protected object involves busy-waiting, then calls are selected for acquiring the execution resource of the protected object in the order in which the busy-wait was initiated; otherwise the FIFO_Spinning policy has no effect. If no Admission_Policy pragma applies to any of the program units in the partition, the admission policy for that partition is implementation defined. 7/5

Discussion: {*AI12-0005-1*} Busy-waiting might be used for protected objects that can be called from tasks running on other processors than the one the protected object is on. It is unnecessary if all of the tasks that can call a protected object are on the same processor as the object; in particular, it would not be used on a monoprocessor. Aspect CPU (see D.16) can be used to ensure that busy-waiting is not needed. 7.a/5

Implementation Note: A possible implementation for this policy would be to apply the abstraction of a ticketing system by assigning two sequence number values to each protected object. One sequence number represents the next available ticket number, and the other sequence number represents the ticket number currently being serviced by the protected object. The next available ticket number is incremented and assigned to a task when the task initiates a busy-wait for acquiring the execution resource associated with the protected object. The ticket number currently being serviced is incremented when a task releases this execution resource. As part of acquiring the execution resource, a task busy-waits until its assigned ticket number equals the protected object's value for the ticket number currently being serviced. While a task busy-waits, it monitors the active priority of the protected object in order to inherit any modifications to the protected object's active priority. 7.b/5

Implementation Permissions

{*AI12-0276-1*} {*AI12-0444-1*} Implementations are allowed to define other admission policies, but are not required to support specifying more than one admission policy per partition. 8/5

Discussion: This rule is in fact redundant, as 10.1.5 allows an implementation to limit the use of configuration pragmas to an empty environment. In that case, there would be no way to have multiple policies in a partition. 8.a/5

Extensions to Ada 2012

{*AI12-0276-1*} Admission Policies and the specific policy FIFO_Spinning are new. 8.b/5

D.5 Dynamic Priorities

{*AI95-00327-01*} {*AI05-0299-1*} [This subclause describes how the priority of an entity can be modified or queried at run time.] 1/3

Wording Changes from Ada 95

{*AI95-00327-01*} {*AI05-0299-1*} This subclause is turned into two subclauses. This subclause introduction is new. 1.a/3

D.5.1 Dynamic Priorities for Tasks

{*AI05-0299-1*} [This subclause describes how the base priority of a task can be modified or queried at run time.] 1/3

Static Semantics

The following language-defined library package exists: 2

```
{AI95-00362-01} {AI12-0241-1} {AI12-0302-1} with System;                    3/5
with Ada.Task_Identification; -- See C.7.1
package Ada.Dynamic_Priorities
    with Preelaborate, Nonblocking, Global => in out synchronized is
```

```
4          procedure Set_Priority(Priority : in System.Any_Priority;
                                  T : in Ada.Task_Identification.Task_Id :=
                                  Ada.Task_Identification.Current_Task);
5          function Get_Priority (T : Ada.Task_Identification.Task_Id :=
                                  Ada.Task_Identification.Current_Task)
                                  return System.Any_Priority;
6       end Ada.Dynamic_Priorities;
```

Dynamic Semantics

7 The procedure Set_Priority sets the base priority of the specified task to the specified Priority value. Set_Priority has no effect if the task is terminated.

8 The function Get_Priority returns T's current base priority. Tasking_Error is raised if the task is terminated.

8.a **Reason:** There is no harm in setting the priority of a terminated task. A previous version of Ada 9X made this a run-time error. However, there is little difference between setting the priority of a terminated task, and setting the priority of a task that is about to terminate very soon; neither case should be an error. Furthermore, the runtime check is not necessarily feasible to implement on all systems, since priority changes might be deferred due to inter-processor communication overhead, so the error might not be detected until after Set_Priority has returned.

8.b However, we wish to allow implementations to avoid storing "extra" information about terminated tasks. Therefore, we make Get_Priority of a terminated task raise an exception; the implementation need not continue to store the priority of a task that has terminated.

9 Program_Error is raised by Set_Priority and Get_Priority if T is equal to Null_Task_Id.

10/2 {*AI95-00188-02*} On a system with a single processor, the setting of the base priority of a task *T* to the new value occurs immediately at the first point when *T* is outside the execution of a protected action.

10.a/2 **Implementation Note:** {*AI95-00188-02*} The priority change is immediate if the target task is on a delay queue or a ready queue outside of a protected action. However, consider when Set_Priority is called by a task T1 to set the priority of T2, if T2 is blocked, waiting on an entry call queued on a protected object, the entry queue needs to be reordered. Since T1 might have a priority that is higher than the ceiling of the protected object, T1 cannot, in general, do the reordering. One way to implement this is to wake T2 up and have T2 do the work. This is similar to the disentangling of queues that needs to happen when a high-priority task aborts a lower-priority task, which might have a call queued on a protected object with a low ceiling. We have an Implementation Permission in D.4 to allow this implementation. We could have required an immediate priority change if on a ready queue during a protected action, but that would have required extra checks for ceiling violations to meet Bounded (Run-Time) Error requirements of D.3 and potentially could cause a protected action to be abandoned in the middle (by raising Program_Error). That seems bad.

10.b **Reason:** A previous version of Ada 9X made it a run-time error for a high-priority task to set the priority of a lower-priority task that has a queued call on a protected object with a low ceiling. This was changed because:

10.c - The check was not feasible to implement on all systems, since priority changes might be deferred due to inter-processor communication overhead. The calling task would continue to execute without finding out whether the operation succeeded or not.

10.d - The runtime check would tend to cause intermittent system failures — how is the caller supposed to know whether the other task happens to have a queued call at any given time? Consider for example an interrupt that needs to trigger a priority change in some task. The interrupt handler could not safely call Set_Priority without knowing exactly what the other task is doing, or without severely restricting the ceilings used in the system. If the interrupt handler wants to hand the job off to a third task whose job is to call Set_Priority, this won't help, because one would normally want the third task to have high priority.

Paragraph 11 was deleted.

Erroneous Execution

12 If any subprogram in this package is called with a parameter T that specifies a task object that no longer exists, the execution of the program is erroneous.

12.a **Ramification:** Note that this rule overrides the above rule saying that Program_Error is raised on Get_Priority of a terminated task. If the task object still exists, and the task is terminated, Get_Priority raises Program_Error. However, if the task object no longer exists, calling Get_Priority causes erroneous execution.

Documentation Requirements

{*AI95-00188-02*} On a multiprocessor, the implementation shall document any conditions that cause the completion of the setting of the priority of a task to be delayed later than what is specified for a single processor. 12.1/2

Documentation Requirement: Any conditions that cause the completion of the setting of the priority of a task to be delayed for a multiprocessor. 12.a.1/2

Metrics

The implementation shall document the following metric: 13

- The execution time of a call to Set_Priority, for the nonpreempting case, in processor clock cycles. This is measured for a call that modifies the priority of a ready task that is not running (which cannot be the calling one), where the new base priority of the affected task is lower than the active priority of the calling task, and the affected task is not on any entry queue and is not executing a protected operation. 14

Documentation Requirement: The metrics for Set_Priority. 14.a/2

NOTE 1 {*AI95-00321-01*} Setting a task's base priority affects task dispatching. First, it can change the task's active priority. Second, under the FIFO_Within_Priorities policy it always causes the task to move to the tail of the ready queue corresponding to its active priority, even if the new base priority is unchanged. 15/2

NOTE 2 Under the priority queuing policy, setting a task's base priority has an effect on a queued entry call if the task is blocked waiting for the call. That is, setting the base priority of a task causes the priority of a queued entry call from that task to be updated and the call to be removed and then reinserted in the entry queue at the new priority (see D.4), unless the call originated from the triggering_statement of an asynchronous_select. 16

NOTE 3 The effect of two or more Set_Priority calls executed in parallel on the same task is defined as executing these calls in some serial order. 17

Proof: This follows from the general reentrancy requirements stated near the beginning of Annex A, “Predefined Language Environment”. 17.a

NOTE 4 {*AI05-0092-1*} {*AI12-0447-1*} The rule for when Tasking_Error is raised for Set_Priority or Get_Priority is different from the rule for when Tasking_Error is raised on an entry call (see 9.5.3). In particular, the priority of a completed or an abnormal task can be queried so long as the task is not yet terminated, and the priority of a task can be set in any task state (including for terminated tasks). 18/5

NOTE 5 {*AI12-0442-1*} Changing the priorities of a set of tasks can be performed by a series of calls to Set_Priority for each task separately. This can be done reliably within a protected operation that has high enough ceiling priority to guarantee that the operation completes without being preempted by any of the affected tasks. 19/5

Extensions to Ada 95

{*AI95-00188-02*} **Amendment Correction:** Priority changes are now required to be done immediately so long as the target task is not on an entry queue. 19.a/2

{*AI95-00362-01*} Dynamic_Priorities is now Preelaborated, so it can be used in preelaborated units. 19.b/2

Wording Changes from Ada 95

{*AI95-00327-01*} {*AI05-0299-1*} This Ada 95 subclause was moved down a level. The paragraph numbers are the same as those for D.5 in Ada 95. 19.c/3

{*AI95-00321-01*} There is no “standard” policy anymore, so that phrase was replaced by the name of a specific policy in the notes. 19.d/2

{*AI95-00327-01*} The bounded error for the priority of a task being higher than the ceiling of an object it is currently in was moved to D.3, so that it applies no matter how the situation arises. 19.e/2

D.5.2 Dynamic Priorities for Protected Objects

{*AI95-00327-01*} {*AI05-0299-1*} This subclause specifies how the priority of a protected object can be modified or queried at run time. 1/3

Static Semantics

2/5 {*AI95-00327-01*} {*AI12-0230-1*} The following attributes are defined for a prefix P that denotes a protected object:

3/2 P'Priority {*AI95-00327-01*} Denotes a non-aliased component of the protected object P. This component is of type System.Any_Priority and its value is the priority of P. P'Priority denotes a variable if and only if P denotes a variable. A reference to this attribute shall appear only within the body of P.

3.1/5 P'Relative_Deadline
{*AI12-0230-1*} Denotes a non-aliased component of the protected object P. This component is of type Ada.Real_Time.Time_Span and its value is the relative deadline of P. P'Relative_Deadline denotes a variable if and only if P denotes a variable. A reference to this attribute shall appear only within the body of P.

4/5 {*AI95-00327-01*} {*AI12-0230-1*} The initial value of the attribute Priority is determined by the initial value of the priority of the protected object (see D.3)[, and can be changed by an assignment]. The initial value of the attribute Relative_Deadline is determined by the initial value of the relative deadline of the protected object (see D.3)[, and can be changed by an assignment].

Dynamic Semantics

5/3 {*AI95-00327-01*} {*AI05-0264-1*} If the locking policy Ceiling_Locking (see D.3) is in effect, then the ceiling priority of a protected object *P* is set to the value of *P*'Priority at the end of each protected action of *P*.

6/3 {*AI95-00445-01*} {*AI05-0229-1*} If the locking policy Ceiling_Locking is in effect, then for a protected object *P* with either an Attach_Handler or Interrupt_Handler aspect specified for one of its procedures, a check is made that the value to be assigned to *P*'Priority is in the range System.Interrupt_Priority. If the check fails, Program_Error is raised.

Metrics

7/2 {*AI95-00327-01*} The implementation shall document the following metric:

8/2 • The difference in execution time of calls to the following procedures in protected object P:

9/2
```
protected P is
   procedure Do_Not_Set_Ceiling (Pr : System.Any_Priority);
   procedure Set_Ceiling (Pr : System.Any_Priority);
end P;
```

10/2
```
protected body P is
   procedure Do_Not_Set_Ceiling (Pr : System.Any_Priority) is
   begin
      null;
   end;
   procedure Set_Ceiling (Pr : System.Any_Priority) is
   begin
      P'Priority := Pr;
   end;
end P;
```

10.a/2 **Documentation Requirement:** The metrics for setting the priority of a protected object.

11/2 NOTE {*AI95-00327-01*} Since P'Priority is a normal variable, the value following an assignment to the attribute immediately reflects the new value even though its impact on the ceiling priority of P is postponed until completion of the protected action in which it is executed.

Extensions to Ada 95

11.a/2 {*AI95-00327-01*} {*AI95-00445-01*} The ability to dynamically change and query the priority of a protected object is new.

Extensions to Ada 2012

{*AI12-0230-1*} The ability to dynamically change and query the relative deadline of a protected object is new. 11.b/5

D.6 Preemptive Abort

{*AI05-0299-1*} [This subclause specifies requirements on the immediacy with which an aborted construct is completed.] 1/3

Dynamic Semantics

On a system with a single processor, an aborted construct is completed immediately at the first point that is outside the execution of an abort-deferred operation. 2

Documentation Requirements

On a multiprocessor, the implementation shall document any conditions that cause the completion of an aborted construct to be delayed later than what is specified for a single processor. 3

This paragraph was deleted. 3.a/2

Documentation Requirement: On a multiprocessor, any conditions that cause the completion of an aborted construct to be delayed later than what is specified for a single processor. 3.b/2

Metrics

The implementation shall document the following metrics: 4

- The execution time, in processor clock cycles, that it takes for an abort_statement to cause the completion of the aborted task. This is measured in a situation where a task T2 preempts task T1 and aborts T1. T1 does not have any finalization code. T2 shall verify that T1 has terminated, by means of the Terminated attribute. 5
- On a multiprocessor, an upper bound in seconds, on the time that the completion of an aborted task can be delayed beyond the point that it is required for a single processor. 6
- {*AI95-00114-01*} An upper bound on the execution time of an asynchronous_select, in processor clock cycles. This is measured between a point immediately before a task T1 executes a protected operation Pr.Set that makes the condition of an entry_barrier Pr.Wait True, and the point where task T2 resumes execution immediately after an entry call to Pr.Wait in an asynchronous_select. T1 preempts T2 while T2 is executing the abortable part, and then blocks itself so that T2 can execute. The execution time of T1 is measured separately, and subtracted. 7/2
- An upper bound on the execution time of an asynchronous_select, in the case that no asynchronous transfer of control takes place. This is measured between a point immediately before a task executes the asynchronous_select with a nonnull abortable part, and the point where the task continues execution immediately after it. The execution time of the abortable part is subtracted. 8

Documentation Requirement: The metrics for aborts. 8.a/2

Implementation Advice

Even though the abort_statement is included in the list of potentially blocking operations (see 9.5.1), it is recommended that this statement be implemented in a way that never requires the task executing the abort_statement to block. 9

Implementation Advice: The abort_statement should not require the task executing the statement to block. 9.a/2

On a multi-processor, the delay associated with aborting a task on another processor should be bounded; the implementation should use periodic polling, if necessary, to achieve this. 10

10.a/2 **Implementation Advice:** On a multi-processor, the delay associated with aborting a task on another processor should be bounded.

11 NOTE 1 Abortion does not change the active or base priority of the aborted task.

12 NOTE 2 Abortion cannot be more immediate than is allowed by the rules for deferral of abortion during finalization and in protected actions.

D.7 Tasking Restrictions

1/3 {*AI05-0299-1*} [This subclause defines restrictions that can be used with a pragma Restrictions (see 13.12) to facilitate the construction of highly efficient tasking run-time systems.]

Static Semantics

1.1/5 {*AI12-0290-1*} {*AI12-0369-1*} A scalar expression within a protected unit is said to be *pure-barrier-eligible* if it is one of the following:

1.2/5 - a static expression;

1.3/5 - a name that statically names (see 4.9) a scalar subcomponent of the immediately enclosing protected unit;

1.4/5 - a Count attribute_reference whose prefix statically denotes an entry declaration of the immediately enclosing unit;

1.5/5 - a call to a predefined relational operator or boolean logical operator (**and**, **or**, **xor**, **not**), where each operand is pure-barrier-eligible;

1.6/5 - a membership test whose *tested*_simple_expression is pure-barrier-eligible, and whose membership_choice_list meets the requirements for a static membership test (see 4.9);

1.7/5 - a short-circuit control form both of whose operands are pure-barrier-eligible;

1.8/5 - a conditional_expression all of whose conditions, *selecting*_expressions, and *dependent*_expressions are pure-barrier-eligible; or

1.9/5 - a pure-barrier-eligible expression enclosed in parentheses.

2 The following *restriction*_identifiers are language defined:

3/3 {*AI05-0013-1*} {*AI05-0216-1*} No_Task_Hierarchy
No task depends on a master other than the library-level master.

3.a/3 **Ramification:** {*AI05-0216-1*} This is equivalent to saying "no task depends on a master other than the master that is the execution of the body of the environment task of the partition", but it is much easier to understand. This is a post-compilation check, which can be checked at compile-time.

3.b/3 {*AI05-0013-1*} This disallows any function returning an object with a task part or coextension, even if called at the library level, as such a task would temporarily depend on a nested master (the master of the return statement), which is disallowed by this restriction.

4/3 {*8652/0042*} {*AI95-00130-01*} {*AI95-00360-01*} {*AI05-0013-1*} No_Nested_Finalization
Objects of a type that needs finalization (see 7.6) are declared only at library level. If an access type does not have library-level accessibility, then there are no allocators of the type where the type determined by the subtype_mark of the subtype_indication or qualified_expression needs finalization.

4.a/1 *This paragraph was deleted.*{*8652/0042*} {*AI95-00130-01*}

4.b/3 **Ramification:** {*AI05-0013-1*} The second sentence prevents the declaration of objects of access types which would require nested finalization. It also prevents the declarations of coextensions that need finalization in a nested scope. The latter cannot be done by preventing the declaration of the objects, as it is not necessarily known if the coextension type needs finalization (it could be a limited view).

{*AI05-0211-1*} No_Abort_Statements 5/3

There are no abort_statements, and there is no use of a name denoting Task_Identification.Abort_Task.

No_Terminate_Alternatives 6

There are no selective_accepts with terminate_alternatives.

No_Task_Allocators 7

There are no allocators for task types or types containing task subcomponents.

{*AI05-0224-1*} In the case of an initialized allocator of an access type whose designated type is class-wide and limited, a check is made that the specific type of the allocated object has no task subcomponents. Program_Error is raised if this check fails. 7.1/3

No_Implicit_Heap_Allocations 8

There are no operations that implicitly require heap storage allocation to be performed by the implementation. The operations that implicitly require heap storage allocation are implementation defined.

Implementation defined: Any operations that implicitly require heap storage allocation. 8.a

{*AI95-00327-01*} No_Dynamic_Priorities 9/2

There are no semantic dependences on the package Dynamic_Priorities, and no occurrences of the attribute Priority.

{*AI95-00305-01*} {*AI95-00394-01*} {*AI05-0013-1*} {*AI05-0211-1*} No_Dynamic_Attachment 10/3

There is no use of a name denoting any of the operations defined in package Interrupts (Is_Reserved, Is_Attached, Current_Handler, Attach_Handler, Exchange_Handler, Detach_Handler, and Reference).

Ramification: {*AI05-0013-1*} This includes 'Access and 'Address of any of these operations, as well as inherited versions of these operations. 10.a/3

{*AI12-0055-1*} No_Dynamic_CPU_Assignment 10.1/4

No task has the CPU aspect specified to be a non-static expression. Each task (including the environment task) that has the CPU aspect specified as Not_A_Specific_CPU will be assigned to a particular implementation-defined CPU. The same is true for the environment task when the CPU aspect is not specified. [Any other task without a CPU aspect will activate and execute on the same processor as its activating task.]

Proof: The processor of a task without a CPU aspect is defined in D.16, and this restriction guarantees that the activator always has a CPU assigned. 10.b/4

Reason: This restriction prevents any migration of tasks. 10.c/4

Ramification: If no CPU aspects are specified, then the program will run on a single CPU, as all of the tasks will be activated directly or indirectly by the environment task, and the rules require the same CPU to be used as the activating task. 10.d/4

Implementation defined: When restriction No_Dynamic_CPU_Assignment applies to a partition, the processor on which a task with a CPU value of a Not_A_Specific_CPU will execute. 10.d.1/4

{*AI95-00305-01*} {*AI05-0013-1*} No_Local_Protected_Objects 10.2/3

Protected objects are declared only at library level.

{*AI95-00297-01*} {*AI05-0013-1*} No_Local_Timing_Events 10.3/3

Timing_Events are declared only at library level.

{*AI95-00305-01*} No_Protected_Type_Allocators 10.4/2

There are no allocators for protected types or types containing protected type subcomponents.

{*AI05-0224-1*} In the case of an initialized allocator of an access type whose designated type is class-wide and limited, a check is made that the specific type of the allocated object has no protected subcomponents. Program_Error is raised if this check fails. 10.5/3

10.6/3 {*AI95-00305-01*} {*AI05-0211-1*} No_Relative_Delay
There are no delay_relative_statements, and there is no use of a name that denotes the Timing_Events.Set_Handler subprogram that has a Time_Span parameter.

10.7/3 {*AI95-00305-01*} No_Requeue_Statements
There are no requeue_statements.

10.8/3 {*AI95-00305-01*} No_Select_Statements
There are no select_statements.

10.9/3 {*AI95-00394-01*} {*AI05-0211-1*} No_Specific_Termination_Handlers
There is no use of a name denoting the Set_Specific_Handler and Specific_Handler subprograms in Task_Termination.

10.10/4 {*AI12-0117-1*} No_Tasks_Unassigned_To_CPU
The CPU aspect is specified for the environment task. No CPU aspect is specified to be statically equal to Not_A_Specific_CPU. If aspect CPU is specified (dynamically) to the value Not_A_Specific_CPU, then Program_Error is raised. If Set_CPU or Delay_Until_And_Set_CPU are called with the CPU parameter equal to Not_A_Specific_CPU, then Program_Error is raised.

10.e/4 **Ramification:** If this restriction is used in a context for which restriction No_Dynamic_CPU_Assignment is in effect, then no runtime check is needed when specifying the CPU aspect. If the restriction is used with the Ravenscar profile, no runtime checks are needed.

10.11/5 {*AI12-0290-1*} Pure_Barriers
The Boolean expression in each protected entry barrier is pure-barrier-eligible.

10.12/5 {*AI95-00305-01*} {*AI05-0013-1*} {*AI12-0369-1*} Simple_Barriers
The Boolean expression in each entry barrier is either a static expression or a name that statically names (see 4.9) a subcomponent of the enclosing protected object.

11 The following *restriction_parameter*_identifiers are language defined:

12 Max_Select_Alternatives
Specifies the maximum number of alternatives in a selective_accept.

13 Max_Task_Entries
Specifies the maximum number of entries per task. The bounds of every entry family of a task unit shall be static, or shall be defined by a discriminant of a subtype whose corresponding bound is static. [A value of zero indicates that no rendezvous are possible.]

14 Max_Protected_Entries
Specifies the maximum number of entries per protected type. The bounds of every entry family of a protected unit shall be static, or shall be defined by a discriminant of a subtype whose corresponding bound is static.

Dynamic Semantics

15/2 {*8652/0076*} {*AI95-00067-01*} {*AI95-00305-01*} The following *restriction*_identifier is language defined:

15.1/2 {*AI95-00305-01*} {*AI95-00394-01*} No_Task_Termination
All tasks are nonterminating. It is implementation-defined what happens if a task attempts to terminate. If there is a fall-back handler (see C.7.3) set for the partition it should be called when the first task attempts to terminate.

15.a.1/2 **Implementation defined:** When restriction No_Task_Termination applies to a partition, what happens when a task terminates.

16 The following *restriction_parameter*_identifiers are language defined:

{*8652/0076*} {*AI95-00067-01*} Max_Storage_At_Blocking 17/1

Specifies the maximum portion [(in storage elements)] of a task's Storage_Size that can be retained by a blocked task. If an implementation chooses to detect a violation of this restriction, Storage_Error should be raised; otherwise, the behavior is implementation defined.

Implementation defined: The behavior when restriction Max_Storage_At_Blocking is violated. 17.a.1/2

{*8652/0076*} {*AI95-00067-01*} Max_Asynchronous_Select_Nesting 18/1

Specifies the maximum dynamic nesting level of asynchronous_selects. A value of zero prevents the use of any asynchronous_select and, if a program contains an asynchronous_-select, it is illegal. If an implementation chooses to detect a violation of this restriction for values other than zero, Storage_Error should be raised; otherwise, the behavior is implementation defined.

Implementation defined: The behavior when restriction Max_Asynchronous_Select_Nesting is violated. 18.a.1/2

{*8652/0076*} {*AI95-00067-01*} Max_Tasks 19/1

Specifies the maximum number of task creations that may be executed over the lifetime of a partition, not counting the creation of the environment task. A value of zero prevents any task creation and, if a program contains a task creation, it is illegal. If an implementation chooses to detect a violation of this restriction, Storage_Error should be raised; otherwise, the behavior is implementation defined.

Ramification: Note that this is not a limit on the number of tasks active at a given time; it is a limit on the total number of task creations that occur. 19.a

Implementation Note: We envision an implementation approach that places TCBs or pointers to them in a fixed-size table, and never reuses table elements. 19.b

Implementation defined: The behavior when restriction Max_Tasks is violated. 19.b.1/2

{*AI95-00305-01*} Max_Entry_Queue_Length 19.1/2

Max_Entry_Queue_Length defines the maximum number of calls that are queued on an entry. Violation of this restriction results in the raising of Program_Error at the point of the call or requeue.

{*AI05-0189-1*} No_Standard_Allocators_After_Elaboration 19.2/3

Specifies that an allocator using a standard storage pool (see 13.11) shall not occur within a parameterless library subprogram, nor within the handled_sequence_of_statements of a task body. For the purposes of this rule, an allocator of a type derived from a formal access type does not use a standard storage pool.

{*AI05-0189-1*} {*AI05-0262-1*} At run time, Storage_Error is raised if an allocator using a standard storage pool is evaluated after the elaboration of the library_items of the partition has completed. 19.3/3

It is implementation defined whether the use of pragma Restrictions results in a reduction in executable program size, storage requirements, or execution time. If possible, the implementation should provide quantitative descriptions of such effects for each restriction. 20

Implementation defined: Whether the use of pragma Restrictions results in a reduction in program code or data size or execution time. 20.a/2

Implementation Advice

When feasible, the implementation should take advantage of the specified restrictions to produce a more efficient implementation. 21

Implementation Advice: When feasible, specified restrictions should be used to produce a more efficient implementation. 21.a/2

NOTE The above Storage_Checks can be suppressed with pragma Suppress. 22

Incompatibilities With Ada 95

22.a/2 {*AI95-00360-01*} **Amendment Correction:** The No_Nested_Finalization is now defined in terms of types that need finalization. These types include a variety of language-defined types that *might* be implemented with a controlled type. If the restriction No_Nested_Finalization (see D.7) applies to the partition, and one of these language-defined types does not have a controlled part, it will not be allowed in local objects in Ada 2005 whereas it would be allowed in original Ada 95. Such code is not portable, as other Ada compilers may have had a controlled part, and thus would be illegal under the restriction.

Extensions to Ada 95

22.b/2 {*AI95-00297-01*} {*AI95-00305-01*} {*AI95-00394-01*} Restrictions No_Dynamic_Attachment, No_Local_Protected_Objects, No_Protected_Type_Allocators, No_Local_Timing_Events, No_Relative_Delay, No_Requeue_Statement, No_Select_Statements, No_Specific_Termination_Handlers, No_Task_Termination, Max_Entry_Queue_Length, and Simple_Barriers are newly added to Ada.

Wording Changes from Ada 95

22.c/2 {*8652/0042*} {*AI95-00130-01*} **Corrigendum:** Clarified that No_Nested_Finalization covered task and protected parts as well.

22.d/2 {*8652/0076*} {*AI95-00067-01*} **Corrigendum:** Changed the description of Max_Tasks and Max_Asynchronous_Select_Nested to eliminate conflicts with the High Integrity Annex (see H.4).

22.e/2 {*AI95-00327-01*} Added using of the new Priority attribute to the restriction No_Dynamic_Priorities.

22.f/2 {*AI95-00394-01*} Restriction No_Asynchronous_Control is now obsolescent.

Incompatibilities With Ada 2005

22.g/3 {*AI05-0013-1*} **Correction:** Changed so that coextensions of types that require nested finalization are also prohibited; this is done by prohibiting allocators rather than objects of specific access types. It seems unlikely that any program depending on this restriction would violate it in this blatant manner, so it is expected that very few programs will be affected by this change.

22.h/3 {*AI05-0211-1*} **Correction:** The restriction No_Relative_Delay was changed to include the Timing_Events routine that uses a relative delay. This means that a program that uses that routine and this restriction will now be rejected. However, such a program violates the spirit and intent of the restriction and as such the program should never have been allowed. Moreover, it is unlikely that any program depending on this restriction would violate it in such an obvious manner, so it is expected that very few programs will be affected by this change.

22.i/3 {*AI05-0211-1*} **Correction:** A number of restrictions were changed from "no calls" on some subprogram to "no use of a name that denotes" that subprogram. This closes a hole where renames, uses as the prefix of 'Access, and the like, would not be rejected by the restriction, possibly allowing backdoor access to the prohibited subprogram. A program that uses one of these restrictions and using such backdoor access will now be rejected; however, it is extremely unlikely that any program that relies on these restrictions would also use an end-run around the restriction, so it is expected that very few programs will be affected by this change.

Extensions to Ada 2005

22.j/3 {*AI05-0189-1*} Restriction No_Standard_Allocators_After_Elaboration is newly added to Ada.

Wording Changes from Ada 2005

22.k/3 {*AI05-0013-1*} {*AI05-0216-1*} **Correction:** Improved the wording of various restrictions to make it clearer that they prohibit things that would otherwise be legal, and to word them as definitions, not Legality Rules;.

22.l/3 {*AI05-0192-1*} **Correction:** Added wording to explain how No_Task_Allocators and No_Protected_Type_Allocators are checked for class-wide types. This might be an extension if the compiler assumed the worst in the past (it is now a runtime check).

Extensions to Ada 2012

22.m/4 {*AI12-0055-1*} **Corrigendum:** Restriction No_Dynamic_CPU_Assignment is newly added to Ada, for use as part of the Ravenscar profile (see D.13).

22.n/4 {*AI12-0117-1*} **Corrigendum:** Restriction No_Tasks_Unassigned_To_CPU is newly added to Ada; it ensures that no task is running on an implementation-defined CPU so that task scheduling can be analyzed.

22.o/5 {*AI12-0290-1*} {*AI12-0369-1*} Restriction Pure_Barriers is newly added to Ada; it allows the Jorvik profile to use more expressive barriers than the Ravenscar profile without having to worry about exceptions or side-effects.

{*AI12-0369-1*} Restriction Simple_Barriers (which is part of the Ravenscar profile) is extended to allow statically named record and array subcomponents; this allows common static analysis patterns to be used with Ravenscar. 22.p/5

D.8 Monotonic Time

{*AI05-0299-1*} [This subclause specifies a high-resolution, monotonic clock package.] 1/3

Static Semantics

The following language-defined library package exists: 2

{*AI12-0241-1*} {*AI12-0302-1*} **package** Ada.Real_Time 3/5
  with Nonblocking, Global => **in out synchronized is**

  type Time **is private**; 4
  Time_First : **constant** Time;
  Time_Last : **constant** Time;
  Time_Unit : **constant** := *implementation-defined-real-number*;

  type Time_Span **is private**; 5
  Time_Span_First : **constant** Time_Span;
  Time_Span_Last : **constant** Time_Span;
  Time_Span_Zero : **constant** Time_Span;
  Time_Span_Unit : **constant** Time_Span;

  Tick : **constant** Time_Span; 6
  function Clock **return** Time;

  function "+" (Left : Time; Right : Time_Span) **return** Time; 7
  function "+" (Left : Time_Span; Right : Time) **return** Time;
  function "-" (Left : Time; Right : Time_Span) **return** Time;
  function "-" (Left : Time; Right : Time) **return** Time_Span;

  function "<" (Left, Right : Time) **return** Boolean; 8
  function "<="(Left, Right : Time) **return** Boolean;
  function ">" (Left, Right : Time) **return** Boolean;
  function ">="(Left, Right : Time) **return** Boolean;

  function "+" (Left, Right : Time_Span) **return** Time_Span; 9
  function "-" (Left, Right : Time_Span) **return** Time_Span;
  function "-" (Right : Time_Span) **return** Time_Span;
  function "*" (Left : Time_Span; Right : Integer) **return** Time_Span;
  function "*" (Left : Integer; Right : Time_Span) **return** Time_Span;
  function "/" (Left, Right : Time_Span) **return** Integer;
  function "/" (Left : Time_Span; Right : Integer) **return** Time_Span;

  function "**abs**"(Right : Time_Span) **return** Time_Span; 10

This paragraph was deleted. 11/1

  function "<" (Left, Right : Time_Span) **return** Boolean; 12
  function "<="(Left, Right : Time_Span) **return** Boolean;
  function ">" (Left, Right : Time_Span) **return** Boolean;
  function ">="(Left, Right : Time_Span) **return** Boolean;

  function To_Duration (TS : Time_Span) **return** Duration; 13
  function To_Time_Span (D : Duration) **return** Time_Span;

{*AI95-00386-01*} **function** Nanoseconds (NS : Integer) **return** Time_Span; 14/2
  function Microseconds (US : Integer) **return** Time_Span;
  function Milliseconds (MS : Integer) **return** Time_Span;
  function Seconds   (S : Integer) **return** Time_Span;
  function Minutes   (M : Integer) **return** Time_Span;

  type Seconds_Count **is range** *implementation-defined*; 15

  procedure Split(T : **in** Time; SC : **out** Seconds_Count; TS : **out** Time_Span); 16
  function Time_Of(SC : Seconds_Count; TS : Time_Span) **return** Time;

private 17
   ... -- *not specified by the language*
end Ada.Real_Time;

This paragraph was deleted. 17.a/2

18 In this Annex, *real time* is defined to be the physical time as observed in the external environment. The type Time is a *time type* as defined by 9.6; [values of this type may be used in a delay_until_statement.] Values of this type represent segments of an ideal time line. The set of values of the type Time corresponds one-to-one with an implementation-defined range of mathematical integers.

18.a **Discussion:** Informally, real time is defined to be the International Atomic Time (TAI) which is monotonic and nondecreasing. We use it here for the purpose of discussing rate of change and monotonic behavior only. It does not imply anything about the absolute value of Real_Time.Clock, or about Real_Time.Time being synchronized with TAI. It is also used for real time in the metrics, for comparison purposes.

18.b **Implementation Note:** The specification of TAI as "real time" does not preclude the use of a simulated TAI clock for simulated execution environments.

19 The Time value I represents the half-open real time interval that starts with E+I*Time_Unit and is limited by E+(I+1)*Time_Unit, where Time_Unit is an implementation-defined real number and E is an unspecified origin point, the *epoch*, that is the same for all values of the type Time. It is not specified by the language whether the time values are synchronized with any standard time reference. [For example, E can correspond to the time of system initialization or it can correspond to the epoch of some time standard.]

19.a **Discussion:** E itself does not have to be a proper time value.

19.b This half-open interval I consists of all real numbers R such that E+I*Time_Unit <= R < E+(I+1)*Time_Unit.

20 Values of the type Time_Span represent length of real time duration. The set of values of this type corresponds one-to-one with an implementation-defined range of mathematical integers. The Time_Span value corresponding to the integer I represents the real-time duration I*Time_Unit.

20.a **Reason:** The purpose of this type is similar to Standard.Duration; the idea is to have a type with a higher resolution.

20.b **Discussion:** We looked at many possible names for this type: Real_Time.Duration, Fine_Duration, Interval, Time_Interval_Length, Time_Measure, and more. Each of these names had some problems, and we've finally settled for Time_Span.

21 Time_First and Time_Last are the smallest and largest values of the Time type, respectively. Similarly, Time_Span_First and Time_Span_Last are the smallest and largest values of the Time_Span type, respectively.

22 A value of type Seconds_Count represents an elapsed time, measured in seconds, since the epoch.

Dynamic Semantics

23 Time_Unit is the smallest amount of real time representable by the Time type; it is expressed in seconds. Time_Span_Unit is the difference between two successive values of the Time type. It is also the smallest positive value of type Time_Span. Time_Unit and Time_Span_Unit represent the same real time duration. A *clock tick* is a real time interval during which the clock value (as observed by calling the Clock function) remains constant. Tick is the average length of such intervals.

24/2 {*AI95-00432-01*} The function To_Duration converts the value TS to a value of type Duration. Similarly, the function To_Time_Span converts the value D to a value of type Time_Span. For To_Duration, the result is rounded to the nearest value of type Duration (away from zero if exactly halfway between two values). If the result is outside the range of Duration, Constraint_Error is raised. For To_Time_Span, the value of D is first rounded to the nearest integral multiple of Time_Unit, away from zero if exactly halfway between two multiples. If the rounded value is outside the range of Time_Span, Constraint_Error is raised. Otherwise, the value is converted to the type Time_Span.

25 To_Duration(Time_Span_Zero) returns 0.0, and To_Time_Span(0.0) returns Time_Span_Zero.

26/2 {*AI95-00386-01*} {*AI95-00432-01*} The functions Nanoseconds, Microseconds, Milliseconds, Seconds, and Minutes convert the input parameter to a value of the type Time_Span. NS, US, MS, S, and M are interpreted as a number of nanoseconds, microseconds, milliseconds, seconds, and minutes respectively.

The input parameter is first converted to seconds and rounded to the nearest integral multiple of Time_Unit, away from zero if exactly halfway between two multiples. If the rounded value is outside the range of Time_Span, Constraint_Error is raised. Otherwise, the rounded value is converted to the type Time_Span.

This paragraph was deleted.{*AI95-00432-01*} 26.a/2

The effects of the operators on Time and Time_Span are as for the operators defined for integer types. 27

Implementation Note: Though time values are modeled by integers, the types Time and Time_Span need not be implemented as integers. 27.a

The function Clock returns the amount of time since the epoch. 28

The effects of the Split and Time_Of operations are defined as follows, treating values of type Time, Time_Span, and Seconds_Count as mathematical integers. The effect of Split(T,SC,TS) is to set SC and TS to values such that T*Time_Unit = SC*1.0 + TS*Time_Unit, and 0.0 <= TS*Time_Unit < 1.0. The value returned by Time_Of(SC,TS) is the value T such that T*Time_Unit = SC*1.0 + TS*Time_Unit. 29

Implementation Requirements

The range of Time values shall be sufficient to uniquely represent the range of real times from program start-up to 50 years later. Tick shall be no greater than 1 millisecond. Time_Unit shall be less than or equal to 20 microseconds. 30

Implementation Note: The required range and accuracy of Time are such that 32-bits worth of seconds and 32-bits worth of ticks in a second could be used as the representation. 30.a

{*AI12-0451-1*} The value of Time_Span_First in seconds shall be no greater than –3600, and the value of Time_Span_Last in seconds shall be no less than 3600. 31/5

Reason: This is equivalent to ± one hour and there is still room for a two-microsecond resolution. 31.a

A *clock jump* is the difference between two successive distinct values of the clock (as observed by calling the Clock function). There shall be no backward clock jumps. 32

Documentation Requirements

The implementation shall document the values of Time_First, Time_Last, Time_Span_First, Time_Span_Last, Time_Span_Unit, and Tick. 33

Documentation Requirement: The values of Time_First, Time_Last, Time_Span_First, Time_Span_Last, Time_Span_Unit, and Tick for package Real_Time. 33.a/2

The implementation shall document the properties of the underlying time base used for the clock and for type Time, such as the range of values supported and any relevant aspects of the underlying hardware or operating system facilities used. 34

Documentation Requirement: The properties of the underlying time base used in package Real_Time. 34.a.1/2

Discussion: If there is an underlying operating system, this might include information about which system call is used to implement the clock. Otherwise, it might include information about which hardware clock is used. 34.a

The implementation shall document whether or not there is any synchronization with external time references, and if such synchronization exists, the sources of synchronization information, the frequency of synchronization, and the synchronization method applied. 35

Documentation Requirement: Any synchronization of package Real_Time with external time references. 35.a.1/2

{*AI05-0299-1*} {*AI12-0439-1*} The implementation shall document any aspects of the external environment that can interfere with the clock behavior as defined in this subclause. 36/5

Documentation Requirement: Any aspects of the external environment that can interfere with package Real_Time. 36.a.1/5

36.a **Discussion:** For example, the implementation is allowed to rely on the time services of an underlying operating system, and this operating system clock can implement time zones or allow the clock to be reset by an operator. This dependence has to be documented.

Metrics

37/3 {*AI05-0299-1*} For the purpose of the metrics defined in this subclause, real time is defined to be the International Atomic Time (TAI).

38 The implementation shall document the following metrics:

39 • An upper bound on the real-time duration of a clock tick. This is a value D such that if t1 and t2 are any real times such that t1 < t2 and $Clock_{t1} = Clock_{t2}$ then t2 – t1 <= D.

40 • An upper bound on the size of a clock jump.

41 • An upper bound on the *drift rate* of Clock with respect to real time. This is a real number D such that

42 $E*(1–D) <= (Clock_{t+E} – Clock_t) <= E*(1+D)$
provided that: $Clock_t + E*(1+D) <= Time\_Last$.

43 • where $Clock_t$ is the value of Clock at time t, and E is a real time duration not less than 24 hours. The value of E used for this metric shall be reported.

43.a **Reason:** This metric is intended to provide a measurement of the long term (cumulative) deviation; therefore, 24 hours is the lower bound on the measurement period. On some implementations, this is also the maximum period, since the language does not require that the range of the type Duration be more than 24 hours. On those implementations that support longer-range Duration, longer measurements should be performed.

44 • An upper bound on the execution time of a call to the Clock function, in processor clock cycles.

45 • Upper bounds on the execution times of the operators of the types Time and Time_Span, in processor clock cycles.

45.a **Implementation Note:** A fast implementation of the Clock function involves repeated reading until you get the same value twice. It is highly improbable that more than three reads will be necessary. Arithmetic on time values should not be significantly slower than 64-bit arithmetic in the underlying machine instruction set.

45.a.1/2 **Documentation Requirement:** The metrics for package Real_Time.

Implementation Permissions

46/5 {*AI12-0444-1*} Implementations targeted to machines with word size smaller than 32 bits may omit support for the full range and granularity of the Time and Time_Span types.

46.a **Discussion:** These requirements are based on machines with a word size of 32 bits.

46.b Since the range and granularity are implementation defined, the supported values need to be documented.

Implementation Advice

47 When appropriate, implementations should provide configuration mechanisms to change the value of Tick.

47.a.1/2 **Implementation Advice:** When appropriate, mechanisms to change the value of Tick should be provided.

47.a **Reason:** This is often needed when the compilation system was originally targeted to a particular processor with a particular interval timer, but the customer uses the same processor with a different interval timer.

47.b **Discussion:** Tick is a deferred constant and not a named number specifically for this purpose.

47.c **Implementation Note:** This can be achieved either by pre-run-time configuration tools, or by having Tick be initialized (in the package private part) by a function call residing in a board specific module.

48 It is recommended that Calendar.Clock and Real_Time.Clock be implemented as transformations of the same time base.

48.a.1/2 **Implementation Advice:** Calendar.Clock and Real_Time.Clock should be transformations of the same time base.

It is recommended that the “best” time base which exists in the underlying system be available to the application through Clock. “Best” may mean highest accuracy or largest range. 49

Implementation Advice: The “best” time base which exists in the underlying system should be available to the application through Real_Time.Clock. 49.a.1/2

NOTE 1 {*AI05-0299-1*} The rules in this subclause do not imply that the implementation can protect the user from operator or installation errors that can result in the clock being set incorrectly. 50/3

NOTE 2 Time_Unit is the granularity of the Time type. In contrast, Tick represents the granularity of Real_Time.Clock. There is no requirement that these be the same. 51

Incompatibilities With Ada 95

{*AI95-00386-01*} {*AI05-0005-1*} Functions Seconds and Minutes are added to Real_Time. If Real_Time is referenced in a use_clause, and an entity *E* with a defining_identifier of Seconds or Minutes is defined in a package that is also referenced in a use_clause, the entity *E* may no longer be use-visible, resulting in errors. This should be rare and is easily fixed if it does occur. 51.a/3

Wording Changes from Ada 95

{*AI95-00432-01*} Added wording explaining how and when many of these functions can raise Constraint_Error. While there always was an intent to raise Constraint_Error if the values did not fit, there never was any wording to that effect, and since Time_Span was a private type, the normal numeric type rules do not apply to it. 51.b/2

D.9 Delay Accuracy

{*AI05-0299-1*} [This subclause specifies performance requirements for the delay_statement. The rules apply both to delay_relative_statement and to delay_until_statement. Similarly, they apply equally to a simple delay_statement and to one which appears in a delay_alternative.] 1/3

Dynamic Semantics

The effect of the delay_statement for Real_Time.Time is defined in terms of Real_Time.Clock: 2

- If C_1 is a value of Clock read before a task executes a delay_relative_statement with duration D, and C_2 is a value of Clock read after the task resumes execution following that delay_statement, then $C_2 - C_1 >= D$. 3
- If C is a value of Clock read after a task resumes execution following a delay_until_statement with Real_Time.Time value T, then C >= T. 4

A simple delay_statement with a negative or zero value for the expiration time does not cause the calling task to be blocked; it is nevertheless a potentially blocking operation (see 9.5.1). 5

{*AI05-0004-1*} When a delay_statement appears in a delay_alternative of a timed_entry_call the selection of the entry call is attempted, regardless of the specified expiration time. When a delay_statement appears in a select_alternative, and a call is queued on one of the open entries, the selection of that entry call proceeds, regardless of the value of the delay expression. 6/3

Ramification: The effect of these requirements is that one has to always attempt a rendezvous, regardless of the value of the delay expression. This can be tested by issuing a timed_entry_call with an expiration time of zero, to an open entry. 6.a

Documentation Requirements

The implementation shall document the minimum value of the delay expression of a delay_relative_statement that causes the task to actually be blocked. 7

Documentation Requirement: The minimum value of the delay expression of a delay_relative_statement that causes a task to actually be blocked. 7.a/2

The implementation shall document the minimum difference between the value of the delay expression of a delay_until_statement and the value of Real_Time.Clock, that causes the task to actually be blocked. 8

8.a/2 *This paragraph was deleted.*

8.b/2 **Documentation Requirement:** The minimum difference between the value of the delay expression of a delay_until_statement and the value of Real_Time.Clock, that causes the task to actually be blocked.

Metrics

9 The implementation shall document the following metrics:

10 • An upper bound on the execution time, in processor clock cycles, of a delay_relative_statement whose requested value of the delay expression is less than or equal to zero.

11 • An upper bound on the execution time, in processor clock cycles, of a delay_until_statement whose requested value of the delay expression is less than or equal to the value of Real_Time.Clock at the time of executing the statement. Similarly, for Calendar.Clock.

12/5 • {*AI12-0445-1*} An upper bound on the *lateness* of a delay_relative_statement, for a positive value of the delay expression, in a situation where the task has sufficient priority to preempt the processor as soon as it becomes ready, and can proceed without waiting for any other execution resources. The upper bound is expressed as a function of the value of the delay expression. The lateness is obtained by subtracting the value of the delay expression from the *actual duration*. The actual duration is measured from a point immediately before a task executes the delay_statement to a point immediately after the task resumes execution following this statement.

13/5 • {*AI12-0445-1*} An upper bound on the lateness of a delay_until_statement, in a situation where the value of the requested expiration time is after the time the task begins executing the statement, the task has sufficient priority to preempt the processor as soon as it becomes ready, and it can proceed without waiting for any other execution resources. The upper bound is expressed as a function of the difference between the requested expiration time and the clock value at the time the statement begins execution. The lateness of a delay_until_statement is obtained by subtracting the requested expiration time from the real time that the task resumes execution following this statement.

13.a/2 **Documentation Requirement:** The metrics for delay statements.

Wording Changes from Ada 83

14.a The rules regarding a timed_entry_call with a very small positive Duration value, have been tightened to always require the check whether the rendezvous is immediately possible.

Wording Changes from Ada 95

14.b/2 {*AI95-00355-01*} The note about “voluntary round-robin’, while still true, has been deleted as potentially confusing as it is describing a different kind of round-robin than is defined by the round-robin dispatching policy.

D.10 Synchronous Task Control

1/3 {*AI05-0299-1*} [This subclause describes a language-defined private semaphore (suspension object), which can be used for *two-stage suspend* operations and as a simple building block for implementing higher-level queues.]

Static Semantics

2 The following language-defined package exists:

3/5
```
{AI95-00362-01} {AI12-0241-1} {AI12-0302-1} package Ada.Synchronous_Task_Control
  with Preelaborate, Nonblocking, Global => in out synchronized is
```

```
{AI12-0241-1}   type Suspension_Object is limited private;                                4/5
  procedure Set_True(S : in out Suspension_Object);
  procedure Set_False(S : in out Suspension_Object);
  function Current_State(S : Suspension_Object) return Boolean;
  procedure Suspend_Until_True(S : in out Suspension_Object)
     with Nonblocking => False;
private
     ... -- not specified by the language
end Ada.Synchronous_Task_Control;
```

The type Suspension_Object is a by-reference type. 5

Implementation Note: {*AI95-00318-02*} The implementation can ensure this by, for example, making the full view an explicitly limited record type. 5.a/2

{*AI05-0168-1*} The following language-defined package exists: 5.1/3

```
{AI05-0168-1} {AI12-0241-1} {AI12-0302-1} with Ada.Real_Time;                            5.2/5
package Ada.Synchronous_Task_Control.EDF
   with Nonblocking, Global => in out synchronized is
   procedure Suspend_Until_True_And_Set_Deadline
      (S  : in out Suspension_Object;
       TS : in     Ada.Real_Time.Time_Span)
      with Nonblocking => False;
end Ada.Synchronous_Task_Control.EDF;
```

Dynamic Semantics

{*AI95-00114-01*} An object of the type Suspension_Object has two visible states: True and False. Upon initialization, its value is set to False. 6/2

Discussion: This object is assumed to be private to the declaring task, i.e. only that task will call Suspend_Until_True on this object, and the count of callers is at most one. Other tasks can, of course, change and query the state of this object. 6.a

{*AI95-00114-01*} The operations Set_True and Set_False are atomic with respect to each other and with respect to Suspend_Until_True; they set the state to True and False respectively. 7/2

Current_State returns the current state of the object. 8

Discussion: This state can change immediately after the operation returns. 8.a

{*AI95-00114-01*} The procedure Suspend_Until_True blocks the calling task until the state of the object S is True; at that point the task becomes ready and the state of the object becomes False. 9/2

{*AI12-0241-1*} Program_Error is raised upon calling Suspend_Until_True if another task is already waiting on that suspension object. 10/5

{*AI05-0168-1*} {*AI05-0269-1*} {*AI12-0241-1*} The procedure Suspend_Until_True_And_Set_Deadline blocks the calling task until the state of the object S is True; at that point the task becomes ready with a deadline of Ada.Real_Time.Clock + TS, and the state of the object becomes False. Program_Error is raised upon calling Suspend_Until_True_And_Set_Deadline if another task is already waiting on that suspension object. 10.1/5

Bounded (Run-Time) Errors

{*AI12-0171-1*} {*AI12-0439-1*} It is a bounded error for two or more tasks to call Suspend_Until_True on the same Suspension_Object concurrently. For each task, Program_Error can be raised, the task can proceed without suspending, or the task can suspend, potentially indefinitely. The state of the suspension object can end up either True or False. 10.2/5

Implementation Requirements

The implementation is required to allow the calling of Set_False and Set_True during any protected action, even one that has its ceiling priority in the Interrupt_Priority range. 11

12/3 NOTE {*AI05-0168-1*} More complex schemes, such as setting the deadline relative to when Set_True is called, can be programmed using a protected object.

Extensions to Ada 95

12.a/2 {*AI95-00362-01*} Synchronous_Task_Control is now Preelaborated, so it can be used in preelaborated units.

Extensions to Ada 2005

12.b/3 {*AI05-0168-1*} Child package Ada.Synchronous_Task_Control.EDF is new.

Wording Changes from Ada 2012

12.c/5 {*AI12-0171-1*} **Correction:** Clarified that Suspend_Until_True should only be called from a single task, and what happens if that is violated.

D.10.1 Synchronous Barriers

1/3 {*AI05-0174-1*} {*AI05-0299-1*} This subclause introduces a language-defined package to synchronously release a group of tasks after the number of blocked tasks reaches a specified count value.

Static Semantics

2/3 {*AI05-0174-1*} The following language-defined library package exists:

3/5
```
{AI12-0241-1} {AI12-0302-1} package Ada.Synchronous_Barriers
   with Preelaborate, Nonblocking, Global => in out synchronized is
```

4/3
```
   subtype Barrier_Limit is Positive range 1 .. implementation-defined;
```

4.a.1/3 **Implementation defined:** The value of Barrier_Limit'Last in Synchronous_Barriers.

5/3
```
   type Synchronous_Barrier (Release_Threshold : Barrier_Limit) is limited private;
```

6/5
```
{AI12-0241-1}    procedure Wait_For_Release (The_Barrier : in out
Synchronous_Barrier;
                                Notified    :    out Boolean)
      with Nonblocking => False;
```

7/3
```
private
   -- not specified by the language
end Ada.Synchronous_Barriers;
```

8/3 {*AI05-0174-1*} Type Synchronous_Barrier needs finalization (see 7.6).

Dynamic Semantics

9/3 {*AI05-0174-1*} Each call to Wait_For_Release blocks the calling task until the number of blocked tasks associated with the Synchronous_Barrier object is equal to Release_Threshold, at which time all blocked tasks are released. Notified is set to True for one of the released tasks, and set to False for all other released tasks.

10/3 {*AI05-0174-1*} The mechanism for determining which task sets Notified to True is implementation defined.

11/3 {*AI05-0174-1*} Once all tasks have been released, a Synchronous_Barrier object may be reused to block another Release_Threshold number of tasks.

12/3 {*AI05-0174-1*} As the first step of the finalization of a Synchronous_Barrier, each blocked task is unblocked and Program_Error is raised at the place of the call to Wait_For_Release.

13/3 {*AI05-0174-1*} It is implementation defined whether an abnormal task which is waiting on a Synchronous_Barrier object is aborted immediately or aborted when the tasks waiting on the object are released.

13.a.1/3 **Implementation defined:** When an aborted task that is waiting on a Synchronous_Barrier is aborted.

This paragraph was deleted.{*AI05-0174-1*} {*AI12-0241-1*} 14/5

Bounded (Run-Time) Errors

{*AI05-0174-1*} It is a bounded error to call Wait_For_Release on a Synchronous_Barrier object after that object is finalized. If the error is detected, Program_Error is raised. Otherwise, the call proceeds normally, which may leave a task blocked forever. 15/3

Extensions to Ada 2005

{*AI05-0174-1*} The package Ada.Synchronous_Barriers is new. 15.a/3

D.11 Asynchronous Task Control

{*AI05-0299-1*} [This subclause introduces a language-defined package to do asynchronous suspend/resume on tasks. It uses a conceptual *held priority* value to represent the task's *held* state.] 1/3

Static Semantics

The following language-defined library package exists: 2

{*AI95-00362-01*} {*AI12-0241-1*} {*AI12-0302-1*} 3/5

```
with Ada.Task_Identification;
package Ada.Asynchronous_Task_Control
  with Preelaborate, Nonblocking, Global => in out synchronized is
  procedure Hold(T : in Ada.Task_Identification.Task_Id);
  procedure Continue(T : in Ada.Task_Identification.Task_Id);
  function Is_Held(T : Ada.Task_Identification.Task_Id)
   return Boolean;
end Ada.Asynchronous_Task_Control;
```

Dynamic Semantics

{*AI95-00357-01*} After the Hold operation has been applied to a task, the task becomes *held*. For each processor there is a conceptual *idle task*, which is always ready. The base priority of the idle task is below System.Any_Priority'First. The *held priority* is a constant of the type Integer whose value is below the base priority of the idle task. 4/2

Discussion: The held state should not be confused with the blocked state as defined in 9.2; the task is still ready. 4.a

{*AI95-00357-01*} For any priority below System.Any_Priority'First, the task dispatching policy is FIFO_Within_Priorities. 4.1/2

To be honest: This applies even if a Task_Dispatching_Policy specifies the policy for all of the priorities of the partition. 4.b/2

Ramification: A task at the held priority never runs, so it is not necessary to implement FIFO_Within_Priorities for systems that have only one policy (such as EDF_Across_Priorities). 4.c/2

{*AI95-00357-01*} The Hold operation sets the state of T to held. For a held task, the active priority is reevaluated as if the base priority of the task were the held priority. 5/2

Ramification: For example, if T is currently inheriting priorities from other sources (e.g. it is executing in a protected action), its active priority does not change, and it continues to execute until it leaves the protected action. 5.a

{*AI95-00357-01*} The Continue operation resets the state of T to not-held; its active priority is then reevaluated as determined by the task dispatching policy associated with its base priority. 6/2

The Is_Held function returns True if and only if T is in the held state. 7

Discussion: Note that the state of T can be changed immediately after Is_Held returns. 7.a

As part of these operations, a check is made that the task identified by T is not terminated. Tasking_Error is raised if the check fails. Program_Error is raised if the value of T is Null_Task_Id. 8

Erroneous Execution

9 If any operation in this package is called with a parameter T that specifies a task object that no longer exists, the execution of the program is erroneous.

Implementation Permissions

10/5 {*AI12-0444-1*} An implementation may omit support for Asynchronous_Task_Control if it is infeasible to support it in the target environment.

10.a **Reason:** A direct implementation of the Asynchronous_Task_Control semantics using priorities is not necessarily efficient enough. Thus, we envision implementations that use some other mechanism to set the "held" state. If there is no other such mechanism, support for Asynchronous_Task_Control might be infeasible, because an implementation in terms of priority would require one idle task per processor. On some systems, programs are not supposed to know how many processors are available, so creating enough idle tasks would be problematic.

11 NOTE 1 It is a consequence of the priority rules that held tasks cannot be dispatched on any processor in a partition (unless they are inheriting priorities) since their priorities are defined to be below the priority of any idle task.

12 NOTE 2 The effect of calling Get_Priority and Set_Priority on a Held task is the same as on any other task.

13 NOTE 3 Calling Hold on a held task or Continue on a non-held task has no effect.

14 NOTE 4 The rules affecting queuing are derived from the above rules, in addition to the normal priority rules:

15 • When a held task is on the ready queue, its priority is so low as to never reach the top of the queue as long as there are other tasks on that queue.

16 • If a task is executing in a protected action, inside a rendezvous, or is inheriting priorities from other sources (e.g. when activated), it continues to execute until it is no longer executing the corresponding construct.

17 • If a task becomes held while waiting (as a caller) for a rendezvous to complete, the active priority of the accepting task is not affected.

18/1 • {*8652/0077*} {*AI95-00111-01*} If a task becomes held while waiting in a selective_accept, and an entry call is issued to one of the open entries, the corresponding accept_alternative executes. When the rendezvous completes, the active priority of the accepting task is lowered to the held priority (unless it is still inheriting from other sources), and the task does not execute until another Continue.

19 • The same holds if the held task is the only task on a protected entry queue whose barrier becomes open. The corresponding entry body executes.

Extensions to Ada 95

19.a/2 {*AI95-00362-01*} Asynchronous_Task_Control is now Preelaborated, so it can be used in preelaborated units.

Wording Changes from Ada 95

19.b/2 {*8652/0077*} {*AI95-00111-01*} **Corrigendum:** Corrected to eliminate the use of the undefined term "accept body".

19.c/2 {*AI95-00357-01*} The description of held tasks was changed to reflect that the calculation of active priorities depends on the dispatching policy of the base priority. Thus, the policy of the held priority was specified in order to avoid surprises (especially when using the EDF policy).

D.12 Other Optimizations and Determinism Rules

1/3 {*AI05-0299-1*} [This subclause describes various requirements for improving the response and determinism in a real-time system.]

Implementation Requirements

2 If the implementation blocks interrupts (see C.3) not as a result of direct user action (e.g. an execution of a protected action) there shall be an upper bound on the duration of this blocking.

2.a **Ramification:** The implementation shall not allow itself to be interrupted when it is in a state where it is unable to support all the language-defined operations permitted in the execution of interrupt handlers. (see 9.5.1).

The implementation shall recognize entry-less protected types. The overhead of acquiring the execution resource of an object of such a type (see 9.5.1) shall be minimized. In particular, there should not be any overhead due to evaluating entry_barrier conditions. 3

Implementation Note: Ideally the overhead should just be a spin-lock. 3.a

Unchecked_Deallocation shall be supported for terminated tasks that are designated by access types, and shall have the effect of releasing all the storage associated with the task. This includes any run-time system or heap storage that has been implicitly allocated for the task by the implementation. 4

Documentation Requirements

The implementation shall document the upper bound on the duration of interrupt blocking caused by the implementation. If this is different for different interrupts or interrupt priority levels, it should be documented for each case. 5

This paragraph was deleted. 5.a/2

Documentation Requirement: The upper bound on the duration of interrupt blocking caused by the implementation. 5.b/2

Metrics

The implementation shall document the following metric: 6

- The overhead associated with obtaining a mutual-exclusive access to an entry-less protected object. This shall be measured in the following way: 7

  For a protected object of the form: 8

```
protected Lock is                                                    9
   procedure Set;
   function Read return Boolean;
private
   Flag : Boolean := False;
end Lock;

protected body Lock is                                               10
   procedure Set is
   begin
      Flag := True;
   end Set;
   function Read return Boolean
   begin
      return Flag;
   end Read;
end Lock;
```

  The execution time, in processor clock cycles, of a call to Set. This shall be measured between the point just before issuing the call, and the point just after the call completes. The function Read shall be called later to verify that Set was indeed called (and not optimized away). The calling task shall have sufficiently high priority as to not be preempted during the measurement period. The protected object shall have sufficiently high ceiling priority to allow the task to call Set. 11

  For a multiprocessor, if supported, the metric shall be reported for the case where no contention (on the execution resource) exists [from tasks executing on other processors]. 12

  Documentation Requirement: The metrics for entry-less protected objects. 12.a/2

D.13 The Ravenscar and Jorvik Profiles

{*AI95-00249-01*} {*AI05-0246-1*} {*AI05-0299-1*} {*AI12-0291-1*} [This subclause defines the Ravenscar and Jorvik profiles.] 1/5

Paragraphs 2 and 3 were moved to 13.12, "Pragma Restrictions and Pragma Profile".

Legality Rules

4/3 {*AI95-00249-01*} {*AI05-0246-1*} {*AI12-0291-1*} The *profile*_identifier Ravenscar and *profile*_identifier Jorvik are usage profiles (see 13.12). For usage profiles Ravenscar and Jorvik, there shall be no *profile*_pragma_argument_associations.

Static Semantics

5/3 {*AI95-00249-01*} {*AI05-0246-1*} The usage profile Ravenscar is equivalent to the following set of pragmas:

6/4
```
{AI95-00249-01} {AI95-00297-01} {AI95-00394-01} {AI05-0171-1} {AI05-0246-1} {AI12-
0055-1} {AI12-0073-1} pragma Task_Dispatching_Policy (FIFO_Within_Priorities);
pragma Locking_Policy (Ceiling_Locking);
pragma Detect_Blocking;
pragma Restrictions (
             No_Abort_Statements,
             No_Dynamic_Attachment,
             No_Dynamic_CPU_Assignment,
             No_Dynamic_Priorities,
             No_Implicit_Heap_Allocations,
             No_Local_Protected_Objects,
             No_Local_Timing_Events,
             No_Protected_Type_Allocators,
             No_Relative_Delay,
             No_Requeue_Statements,
             No_Select_Statements,
             No_Specific_Termination_Handlers,
             No_Task_Allocators,
             No_Task_Hierarchy,
             No_Task_Termination,
             Simple_Barriers,
             Max_Entry_Queue_Length => 1,
             Max_Protected_Entries => 1,
             Max_Task_Entries => 0,
             No_Dependence => Ada.Asynchronous_Task_Control,
             No_Dependence => Ada.Calendar,
             No_Dependence => Ada.Execution_Time.Group_Budgets,
             No_Dependence => Ada.Execution_Time.Timers,
             No_Dependence => Ada.Synchronous_Barriers,
             No_Dependence => Ada.Task_Attributes,
             No_Dependence => System.Multiprocessors.Dispatching_Domains);
```

6.a/3 **Discussion:** The Ravenscar profile is named for the location of the meeting that defined its initial version. The name is now in widespread use, so we stick with existing practice, rather than using a more descriptive name.

6.b/5 The Jorvik profile is named for the city of York, UK, near where the Ravenscar profile was created. "Jorvik" (pronounced "Yorvick") was the Viking name for York.

6.1/5 {*AI12-0291-1*} The usage profile Jorvik is equivalent to the following set of pragmas:

```
{AI12-0291-1} pragma Task_Dispatching_Policy (FIFO_Within_Priorities);
pragma Locking_Policy (Ceiling_Locking);
pragma Detect_Blocking;
pragma Restrictions (
             No_Abort_Statements,
             No_Dynamic_Attachment,
             No_Dynamic_CPU_Assignment,
             No_Dynamic_Priorities,
             No_Local_Protected_Objects,
             No_Local_Timing_Events,
             No_Protected_Type_Allocators,
             No_Requeue_Statements,
             No_Select_Statements,
             No_Specific_Termination_Handlers,
             No_Task_Allocators,
             No_Task_Hierarchy,
             No_Task_Termination,
             Pure_Barriers,
             Max_Task_Entries => 0,
             No_Dependence => Ada.Asynchronous_Task_Control,
             No_Dependence => Ada.Execution_Time.Group_Budgets,
             No_Dependence => Ada.Execution_Time.Timers,
             No_Dependence => Ada.Task_Attributes,
             No_Dependence => System.Multiprocessors.Dispatching_Domains);
```

6.2/5

Discussion: The Jorvik profile removes a number of restrictions from the Ravenscar profile to allow additional applications to benefit from predictability and low overhead. Specifically, the following restrictions are removed: 6.c/5

No_Implicit_Heap_Allocations 6.d/5
No_Relative_Delay
Max_Entry_Queue_Length => 1
Max_Protected_Entries => 1
No_Dependence => Ada.Calendar
No_Dependence => Ada.Synchronous_Barriers

Jorvik also replaces restriction Simple_Barriers with Pure_Barriers (a weaker requirement than Simple_Barriers). 6.e/5

Paragraph 7 and 8 were deleted.

Implementation Advice

{*AI05-0171-1*} {*AI12-0291-1*} On a multiprocessor system, an implementation should support a fully partitioned approach if one of these profiles is specified. Each processor should have separate and disjoint ready queues. 9/5

Implementation Advice: On a multiprocessor system, each processor should have a separate and disjoint ready queue. 9.a.1/3

NOTE 1 {*AI95-00249-01*} {*AI05-0246-1*} {*AI12-0291-1*} For the Ravenscar profile, the effect of the restriction Max_Entry_Queue_Length => 1 applies only to protected entry queues due to the accompanying restriction Max_Task_Entries => 0. The restriction Max_Entry_Queue_Length is not applied by the Jorvik profile. 10/5

NOTE 2 {*AI12-0055-1*} {*AI12-0291-1*} When the Ravenscar or Jorvik profile is in effect (via the effect of the No_Dynamic_CPU_Assignment restriction), all of the tasks in the partition will execute on a single CPU unless the programmer explicitly uses aspect CPU to specify the CPU assignments for tasks. The use of multiple CPUs requires care, as many guarantees of single CPU scheduling no longer apply. 11/5

NOTE 3 {*AI12-0055-1*} {*AI12-0291-1*} It is not recommended to specify the CPU of a task to be Not_A_Specific_CPU when the Ravenscar or Jorvik profile is in effect. How a partition executes strongly depends on the assignment of tasks to CPUs. 12/5

NOTE 4 {*AI12-0291-1*} Any unit that meets the requirements of the Ravenscar profile also meets the requirements of the Jorvik profile. 13/5

Extensions to Ada 95

{*AI95-00249-01*} {*AI05-0246-1*} The Ravenscar profile is new; it was moved here by Ada 2012. 13.a/3

Wording Changes from Ada 2005

{*AI05-0171-1*} How Ravenscar behaves on a multiprocessor system is now defined. 13.b/3

Incompatibilities With Ada 2012

13.c/4 {*AI05-0073-1*} **Corrigendum:** The Ravenscar profile no longer allows the use of package Synchronous_Barriers, as this package violates the fundamental Ravenscar requirement that each waiting point can only block (and release) a single task. This is incompatible with the published Ada 2012 standard, but it is unlikely that any existing Ravenscar runtime ever usefully supported barriers.

13.d/4 {*AI05-0055-1*} **Corrigendum:**The Ravenscar profile (via the effect of the new restriction No_Dynamic_CPU_Assignment) no longer allows setting the CPU aspect of a task to a non-static value. While this was allowed, an implementation would have had to come up with a creative interpretation of the Ada 2012 requirement to define the association of tasks to processors statically. As such, the new check is more likely to catch bugs than break a working program.

Extensions to Ada 2012

13.e/5 {*AI12-0291-1*} The Jorvik profile is new.

D.14 Execution Time

1/3 {*AI95-00307-01*} {*AI05-0299-1*} This subclause describes a language-defined package to measure execution time.

Static Semantics

2/2 {*AI95-00307-01*} The following language-defined library package exists:

3/5
```
{AI12-0241-1} {AI12-0302-1} with Ada.Task_Identification;
with Ada.Real_Time; use Ada.Real_Time;
package Ada.Execution_Time
   with Nonblocking, Global => in out synchronized is
```

4/2
```
   type CPU_Time is private;
   CPU_Time_First : constant CPU_Time;
   CPU_Time_Last  : constant CPU_Time;
   CPU_Time_Unit  : constant := implementation-defined-real-number;
   CPU_Tick : constant Time_Span;
```

5/2
```
   function Clock
     (T : Ada.Task_Identification.Task_Id
          := Ada.Task_Identification.Current_Task)
     return CPU_Time;
```

6/2
```
   function "+"  (Left : CPU_Time; Right : Time_Span) return CPU_Time;
   function "+"  (Left : Time_Span; Right : CPU_Time) return CPU_Time;
   function "-"  (Left : CPU_Time; Right : Time_Span) return CPU_Time;
   function "-"  (Left : CPU_Time; Right : CPU_Time)  return Time_Span;
```

7/2
```
   function "<"  (Left, Right : CPU_Time) return Boolean;
   function "<=" (Left, Right : CPU_Time) return Boolean;
   function ">"  (Left, Right : CPU_Time) return Boolean;
   function ">=" (Left, Right : CPU_Time) return Boolean;
```

8/2
```
   procedure Split
     (T : in CPU_Time; SC : out Seconds_Count; TS : out Time_Span);
```

9/2
```
   function Time_Of (SC : Seconds_Count;
                     TS : Time_Span := Time_Span_Zero) return CPU_Time;
```

9.1/3
```
   {AI05-0170-1}    Interrupt_Clocks_Supported : constant Boolean := implementation-defined;
```

9.2/3
```
   {AI05-0170-1}    Separate_Interrupt_Clocks_Supported : constant Boolean :=
        implementation-defined;
```

9.3/3
```
   {AI05-0170-1}    function Clock_For_Interrupts return CPU_Time;
```

10/2
```
   private
      ... -- not specified by the language
   end Ada.Execution_Time;
```

11/3 {*AI95-00307-01*} {*AI05-0170-1*} {*AI05-0269-1*} The *execution time* or CPU time of a given task is defined as the time spent by the system executing that task, including the time spent executing run-time or

system services on its behalf. The mechanism used to measure execution time is implementation defined. The Boolean constant Interrupt_Clocks_Supported is set to True if the implementation separately accounts for the execution time of interrupt handlers. If it is set to False it is implementation defined which task, if any, is charged the execution time that is consumed by interrupt handlers. The Boolean constant Separate_Interrupt_Clocks_Supported is set to True if the implementation separately accounts for the execution time of individual interrupt handlers (see D.14.3).

Discussion: The implementation-defined properties above and of the values declared in the package are repeated in Documentation Requirements, so we don't mark them as implementation-defined. 11.a/2

{*AI95-00307-01*} The type CPU_Time represents the execution time of a task. The set of values of this type corresponds one-to-one with an implementation-defined range of mathematical integers. 12/2

{*AI95-00307-01*} The CPU_Time value I represents the half-open execution-time interval that starts with I*CPU_Time_Unit and is limited by (I+1)*CPU_Time_Unit, where CPU_Time_Unit is an implementation-defined real number. For each task, the execution time value is set to zero at the creation of the task. 13/2

Ramification: Since it is implementation-defined which task is charged execution time for system services, the execution time value may become nonzero even before the start of the activation of the task. 13.a/2

{*AI95-00307-01*} CPU_Time_First and CPU_Time_Last are the smallest and largest values of the CPU_Time type, respectively. 14/2

{*AI05-0170-1*} The execution time value for the function Clock_For_Interrupts is initialized to zero. 14.1/3

Dynamic Semantics

{*AI95-00307-01*} CPU_Time_Unit is the smallest amount of execution time representable by the CPU_Time type; it is expressed in seconds. A *CPU clock tick* is an execution time interval during which the clock value (as observed by calling the Clock function) remains constant. CPU_Tick is the average length of such intervals. 15/2

{*AI95-00307-01*} The effects of the operators on CPU_Time and Time_Span are as for the operators defined for integer types. 16/2

{*AI95-00307-01*} The function Clock returns the current execution time of the task identified by T; Tasking_Error is raised if that task has terminated; Program_Error is raised if the value of T is Task_Identification.Null_Task_Id. 17/2

{*AI95-00307-01*} The effects of the Split and Time_Of operations are defined as follows, treating values of type CPU_Time, Time_Span, and Seconds_Count as mathematical integers. The effect of Split (T, SC, TS) is to set SC and TS to values such that T*CPU_Time_Unit = SC*1.0 + TS*CPU_Time_Unit, and 0.0 <= TS*CPU_Time_Unit < 1.0. The value returned by Time_Of(SC,TS) is the execution-time value T such that T*CPU_Time_Unit=SC*1.0 + TS*CPU_Time_Unit. 18/2

{*AI05-0170-1*} The function Clock_For_Interrupts returns the total cumulative time spent executing within all interrupt handlers. This time is not allocated to any task execution time clock. If Interrupt_Clocks_Supported is set to False the function raises Program_Error. 18.1/3

Erroneous Execution

{*AI95-00307-01*} For a call of Clock, if the task identified by T no longer exists, the execution of the program is erroneous. 19/2

Implementation Requirements

20/2 {*AI95-00307-01*} The range of CPU_Time values shall be sufficient to uniquely represent the range of execution times from the task start-up to 50 years of execution time later. CPU_Tick shall be no greater than 1 millisecond.

Documentation Requirements

21/2 {*AI95-00307-01*} The implementation shall document the values of CPU_Time_First, CPU_Time_Last, CPU_Time_Unit, and CPU_Tick.

21.a/2 **Documentation Requirement:** The values of CPU_Time_First, CPU_Time_Last, CPU_Time_Unit, and CPU_Tick of package Execution_Time.

22/2 {*AI95-00307-01*} The implementation shall document the properties of the underlying mechanism used to measure execution times, such as the range of values supported and any relevant aspects of the underlying hardware or operating system facilities used.

22.a/3 **Documentation Requirement:** The properties of the mechanism used to implement package Execution_Time, including the values of the constants defined in the package.

Metrics

23/2 {*AI95-00307-01*} The implementation shall document the following metrics:

24/2 • An upper bound on the execution-time duration of a clock tick. This is a value D such that if t1 and t2 are any execution times of a given task such that t1 < t2 and $Clock_{t1}$ = $Clock_{t2}$ then t2 – t1 <= D.

25/2 • An upper bound on the size of a clock jump. A clock jump is the difference between two successive distinct values of an execution-time clock (as observed by calling the Clock function with the same Task_Id).

26/2 • An upper bound on the execution time of a call to the Clock function, in processor clock cycles.

27/2 • Upper bounds on the execution times of the operators of the type CPU_Time, in processor clock cycles.

27.a/2 **Documentation Requirement:** The metrics for execution time.

Implementation Permissions

28/5 {*AI95-00307-01*} {*AI12-0444-1*} Implementations targeted to machines with word size smaller than 32 bits may omit support for the full range and granularity of the CPU_Time type.

Implementation Advice

29/2 {*AI95-00307-01*} When appropriate, implementations should provide configuration mechanisms to change the value of CPU_Tick.

29.a/2 **Implementation Advice:** When appropriate, implementations should provide configuration mechanisms to change the value of Execution_Time.CPU_Tick.

Extensions to Ada 95

29.b/2 {*AI95-00307-01*} The package Execution_Time is new.

Incompatibilities With Ada 2005

29.c/3 {*AI05-0170-1*} Function Clock_For_Interrupts, and constants Interrupt_Clocks_Supported and Separate_Interrupt_Clocks_Supported are added to Execution_Time. If Execution_Time is referenced in a use_clause, and an entity *E* with a defining_identifier of one of the added entities is defined in a package that is also referenced in a use_clause, the entity *E* may no longer be use-visible, resulting in errors. This should be rare and is easily fixed if it does occur.

Wording Changes from Ada 2005

{*AI05-0170-1*} If Interrupt_Clocks_Supported is True, it is now possible to determine the execution time of interrupt handlers. This is not an inconsistency, as not charging any task for such time was a legitimate implementation for Ada 2005. 29.d/3

D.14.1 Execution Time Timers

{*AI95-00307-01*} {*AI05-0299-1*} This subclause describes a language-defined package that provides a facility for calling a handler when a task has used a defined amount of CPU time. 1/3

Static Semantics

{*AI95-00307-01*} The following language-defined library package exists: 2/2

```
{AI12-0241-1} {AI12-0302-1} with System;                                    3/5
package Ada.Execution_Time.Timers
   with Nonblocking, Global => in out synchronized is

   type Timer (T : not null access constant                                 4/2
                   Ada.Task_Identification.Task_Id) is
      tagged limited private;

{AI12-0241-1}    type Timer_Handler is                                      5/5
      access protected procedure (TM : in out Timer)
      with Nonblocking => False;

   Min_Handler_Ceiling : constant System.Any_Priority :=                    6/2
   implementation-defined;

   procedure Set_Handler (TM      : in out Timer;                           7/2
                          In_Time : in Time_Span;
                          Handler : in Timer_Handler);
   procedure Set_Handler (TM      : in out Timer;
                          At_Time : in CPU_Time;
                          Handler : in Timer_Handler);
   function Current_Handler (TM : Timer) return Timer_Handler;
   procedure Cancel_Handler (TM        : in out Timer;
                             Cancelled :    out Boolean);

   function Time_Remaining (TM : Timer) return Time_Span;                   8/2

   Timer_Resource_Error : exception;                                        9/2

private                                                                     10/2
   ... -- not specified by the language
end Ada.Execution_Time.Timers;
```

{*AI95-00307-01*} The type Timer represents an execution-time event for a single task and is capable of detecting execution-time overruns. The access discriminant T identifies the task concerned. The type Timer needs finalization (see 7.6). 11/2

{*AI95-00307-01*} An object of type Timer is said to be *set* if it is associated with a nonnull value of type Timer_Handler and *cleared* otherwise. All Timer objects are initially cleared. 12/2

{*AI95-00307-01*} The type Timer_Handler identifies a protected procedure to be executed by the implementation when the timer expires. Such a protected procedure is called a *handler*. 13/2

Discussion: Type Timer is tagged. This makes it possible to share a handler between several events. In simple cases, 'Access can be used to compare the parameter with a specific timer object (this works because a tagged type is a by-reference type). In more complex cases, a type extension of type Timer can be declared; a double type conversion can be used to access the extension data. An example of how this can be done can be found for the similar type Timing_Event, see D.15. 13.a/2

Dynamic Semantics

{*AI95-00307-01*} When a Timer object is created, or upon the first call of a Set_Handler procedure with the timer as parameter, the resources required to operate an execution-time timer based on the associated 14/2

execution-time clock are allocated and initialized. If this operation would exceed the available resources, Timer_Resource_Error is raised.

15/3 {*AI95-00307-01*} {*AI05-0264-1*} The procedures Set_Handler associate the handler Handler with the timer TM: if Handler is **null**, the timer is cleared; otherwise, it is set. The first procedure Set_Handler loads the timer TM with an interval specified by the Time_Span parameter. In this mode, the timer TM *expires* when the execution time of the task identified by TM.T.**all** has increased by In_Time; if In_Time is less than or equal to zero, the timer expires immediately. The second procedure Set_Handler loads the timer TM with the absolute value specified by At_Time. In this mode, the timer TM expires when the execution time of the task identified by TM.T.**all** reaches At_Time; if the value of At_Time has already been reached when Set_Handler is called, the timer expires immediately.

15.a/2 **Implementation Note:** Since an access-to-constant can designate a variable, the Task_Id value designated by the discriminant of a Timer object can be changed after the object is created. Thus, an implementation cannot use the value of the Task_Id other than where this Reference Manual specifies. For instance, the Task_Id should be read when the timer is set, but it should not be used when the timer expires (as it may designate a different task at that point).

16/2 {*AI95-00307-01*} A call of a procedure Set_Handler for a timer that is already set replaces the handler and the (absolute or relative) execution time; if Handler is not **null**, the timer remains set.

17/2 {*AI95-00307-01*} When a timer expires, the associated handler is executed, passing the timer as parameter. The initial action of the execution of the handler is to clear the event.

18/3 {*AI95-00307-01*} {*AI05-0264-1*} The function Current_Handler returns the handler associated with the timer TM if that timer is set; otherwise, it returns **null**.

19/3 {*AI95-00307-01*} {*AI05-0264-1*} The procedure Cancel_Handler clears the timer if it is set. Cancelled is assigned True if the timer was set prior to it being cleared; otherwise, it is assigned False.

20/3 {*AI95-00307-01*} {*AI05-0264-1*} The function Time_Remaining returns the execution time interval that remains until the timer TM would expire, if that timer is set; otherwise, it returns Time_Span_Zero.

21/2 {*AI95-00307-01*} The constant Min_Handler_Ceiling is the minimum ceiling priority required for a protected object with a handler to ensure that no ceiling violation will occur when that handler is invoked.

22/2 {*AI95-00307-01*} As part of the finalization of an object of type Timer, the timer is cleared.

23/2 {*AI95-00307-01*} For all the subprograms defined in this package, Tasking_Error is raised if the task identified by TM.T.**all** has terminated, and Program_Error is raised if the value of TM.T.**all** is Task_Identification.Null_Task_Id.

24/2 {*AI95-00307-01*} An exception propagated from a handler invoked as part of the expiration of a timer has no effect.

Erroneous Execution

25/2 {*AI95-00307-01*} For a call of any of the subprograms defined in this package, if the task identified by TM.T.**all** no longer exists, the execution of the program is erroneous.

Implementation Requirements

26/2 {*AI95-00307-01*} For a given Timer object, the implementation shall perform the operations declared in this package atomically with respect to any of these operations on the same Timer object. The replacement of a handler by a call of Set_Handler shall be performed atomically with respect to the execution of the handler.

26.a/2 **Reason:** This prevents various race conditions. In particular it ensures that if an event occurs when Set_Handler is changing the handler then either the new or old handler is executed in response to the appropriate event. It is never possible for a new handler to be executed in response to an old event

{*AI95-00307-01*} When an object of type Timer is finalized, the system resources used by the timer shall be deallocated. 27/2

Implementation Permissions

{*AI95-00307-01*} {*AI05-0264-1*} Implementations may limit the number of timers that can be defined for each task. If this limit is exceeded, then Timer_Resource_Error is raised. 28/3

NOTE {*AI95-00307-01*} A Timer_Handler can be associated with several Timer objects. 29/2

Extensions to Ada 95

{*AI95-00307-01*} The package Execution_Time.Timers is new. 29.a/2

D.14.2 Group Execution Time Budgets

{*AI95-00354-01*} {*AI05-0299-1*} This subclause describes a language-defined package to assign execution time budgets to groups of tasks. 1/3

Static Semantics

{*AI95-00354-01*} The following language-defined library package exists: 2/2

```
{AI05-0169-1} {AI12-0241-1} {AI12-0302-1} with System;                                    3/5
with System.Multiprocessors;
package Ada.Execution_Time.Group_Budgets
  with Nonblocking, Global => in out synchronized is

{AI05-0092-1} {AI05-0169-1}   type Group_Budget(CPU : System.Multiprocessors.CPU :=       4/3
                                System.Multiprocessors.CPU'First)
    is tagged limited private;

{AI12-0241-1}   type Group_Budget_Handler is access                                       5/5
       protected procedure (GB : in out Group_Budget)
       with Nonblocking => False;

  type Task_Array is array (Positive range <>) of                                         6/2
                                Ada.Task_Identification.Task_Id;

  Min_Handler_Ceiling : constant System.Any_Priority :=                                   7/2
    implementation-defined;
```

Implementation defined: The value of Min_Handler_Ceiling in Execution_Time.Group_Budgets. 7.a.1/3

```
  procedure Add_Task (GB : in out Group_Budget;                                           8/2
                      T  : in Ada.Task_Identification.Task_Id);
  procedure Remove_Task (GB: in out Group_Budget;
                         T  : in Ada.Task_Identification.Task_Id);
  function Is_Member (GB : Group_Budget;
                      T : Ada.Task_Identification.Task_Id) return Boolean;
  function Is_A_Group_Member
     (T : Ada.Task_Identification.Task_Id) return Boolean;
  function Members (GB : Group_Budget) return Task_Array;

  procedure Replenish (GB : in out Group_Budget; To : in Time_Span);                      9/2
  procedure Add (GB : in out Group_Budget; Interval : in Time_Span);
  function Budget_Has_Expired (GB : Group_Budget) return Boolean;
  function Budget_Remaining (GB : Group_Budget) return Time_Span;

  procedure Set_Handler (GB      : in out Group_Budget;                                  10/2
                         Handler : in Group_Budget_Handler);
  function Current_Handler (GB : Group_Budget)
     return Group_Budget_Handler;
  procedure Cancel_Handler (GB        : in out Group_Budget;
                            Cancelled : out Boolean);

  Group_Budget_Error : exception;                                                         11/2
```

12/2
```
private
   -- not specified by the language
end Ada.Execution_Time.Group_Budgets;
```

13/2 {*AI95-00354-01*} The type Group_Budget represents an execution time budget to be used by a group of tasks. The type Group_Budget needs finalization (see 7.6). A task can belong to at most one group. Tasks of any priority can be added to a group.

14/2 {*AI95-00354-01*} An object of type Group_Budget has an associated nonnegative value of type Time_Span known as its *budget*, which is initially Time_Span_Zero. The type Group_Budget_Handler identifies a protected procedure to be executed by the implementation when the budget is *exhausted*, that is, reaches zero. Such a protected procedure is called a *handler*.

15/2 {*AI95-00354-01*} An object of type Group_Budget also includes a handler, which is a value of type Group_Budget_Handler. The handler of the object is said to be *set* if it is not null and *cleared* otherwise. The handler of all Group_Budget objects is initially cleared.

15.a/2 **Discussion:** Type Group_Budget is tagged. This makes it possible to share a handler between several events. In simple cases, 'Access can be used to compare the parameter with a specific group budget object (this works because a tagged type is a by-reference type). In more complex cases, a type extension of type Group_Budget can be declared; a double type conversion can be used to access the extension data. An example of how this can be done can be found for the similar type Timing_Event, see D.15.

Dynamic Semantics

16/2 {*AI95-00354-01*} The procedure Add_Task adds the task identified by T to the group GB; if that task is already a member of some other group, Group_Budget_Error is raised.

17/2 {*AI95-00354-01*} The procedure Remove_Task removes the task identified by T from the group GB; if that task is not a member of the group GB, Group_Budget_Error is raised. After successful execution of this procedure, the task is no longer a member of any group.

18/3 {*AI95-00354-01*} {*AI05-0264-1*} The function Is_Member returns True if the task identified by T is a member of the group GB; otherwise, it returns False.

19/3 {*AI95-00354-01*} {*AI05-0264-1*} The function Is_A_Group_Member returns True if the task identified by T is a member of some group; otherwise, it returns False.

20/2 {*AI95-00354-01*} The function Members returns an array of values of type Task_Identification.Task_Id identifying the members of the group GB. The order of the components of the array is unspecified.

21/3 {*AI95-00354-01*} {*AI05-0092-1*} {*AI05-0169-1*} The procedure Replenish loads the group budget GB with To as the Time_Span value. The exception Group_Budget_Error is raised if the Time_Span value To is nonpositive. Any execution on CPU of any member of the group of tasks results in the budget counting down, unless exhausted. When the budget becomes exhausted (reaches Time_Span_Zero), the associated handler is executed if the handler of group budget GB is set. Nevertheless, the tasks continue to execute.

22/2 {*AI95-00354-01*} The procedure Add modifies the budget of the group GB. A positive value for Interval increases the budget. A negative value for Interval reduces the budget, but never below Time_Span_Zero. A zero value for Interval has no effect. A call of procedure Add that results in the value of the budget going to Time_Span_Zero causes the associated handler to be executed if the handler of the group budget GB is set.

23/3 {*AI95-00354-01*} {*AI05-0264-1*} The function Budget_Has_Expired returns True if the budget of group GB is exhausted (equal to Time_Span_Zero); otherwise, it returns False.

24/2 {*AI95-00354-01*} The function Budget_Remaining returns the remaining budget for the group GB. If the budget is exhausted it returns Time_Span_Zero. This is the minimum value for a budget.

{*AI95-00354-01*} {*AI05-0264-1*} The procedure Set_Handler associates the handler Handler with the Group_Budget GB: if Handler is **null**, the handler of Group_Budget is cleared; otherwise, it is set. 25/3

{*AI95-00354-01*} A call of Set_Handler for a Group_Budget that already has a handler set replaces the handler; if Handler is not **null**, the handler for Group_Budget remains set. 26/2

{*AI95-00354-01*} {*AI05-0264-1*} The function Current_Handler returns the handler associated with the group budget GB if the handler for that group budget is set; otherwise, it returns **null**. 27/3

{*AI95-00354-01*} {*AI05-0264-1*} The procedure Cancel_Handler clears the handler for the group budget if it is set. Cancelled is assigned True if the handler for the group budget was set prior to it being cleared; otherwise, it is assigned False. 28/3

{*AI95-00354-01*} The constant Min_Handler_Ceiling is the minimum ceiling priority required for a protected object with a handler to ensure that no ceiling violation will occur when that handler is invoked. 29/2

{*AI95-00354-01*} The precision of the accounting of task execution time to a Group_Budget is the same as that defined for execution-time clocks from the parent package. 30/2

{*AI95-00354-01*} As part of the finalization of an object of type Group_Budget all member tasks are removed from the group identified by that object. 31/2

{*AI95-00354-01*} {*AI05-0264-1*} If a task is a member of a Group_Budget when it terminates, then as part of the finalization of the task it is removed from the group. 32/3

{*AI95-00354-01*} For all the operations defined in this package, Tasking_Error is raised if the task identified by T has terminated, and Program_Error is raised if the value of T is Task_Identification.Null_Task_Id. 33/2

{*AI95-00354-01*} An exception propagated from a handler invoked when the budget of a group of tasks becomes exhausted has no effect. 34/2

Erroneous Execution

{*AI95-00354-01*} For a call of any of the subprograms defined in this package, if the task identified by T no longer exists, the execution of the program is erroneous. 35/2

Implementation Requirements

{*AI95-00354-01*} For a given Group_Budget object, the implementation shall perform the operations declared in this package atomically with respect to any of these operations on the same Group_Budget object. The replacement of a handler, by a call of Set_Handler, shall be performed atomically with respect to the execution of the handler. 36/2

Reason: This prevents various race conditions. In particular it ensures that if the budget is exhausted when Set_Handler is changing the handler then either the new or old handler is executed and the exhausting event is not lost. 36.a/2

NOTE 1 {*AI95-00354-01*} Clearing or setting of the handler of a group budget does not change the current value of the budget. Exhaustion or loading of a budget does not change whether the handler of the group budget is set or cleared. 37/2

NOTE 2 {*AI95-00354-01*} A Group_Budget_Handler can be associated with several Group_Budget objects. 38/2

Extensions to Ada 95

{*AI95-00354-01*} The package Execution_Time.Group_Budgets is new. 38.a/2

Inconsistencies With Ada 2005

{*AI05-0169-1*} A Group_Budget is now defined to work on a single processor. If an implementation managed to make this package work for programs running on a multiprocessor system, and a program depends on that fact, it could fail when ported to Ada 2012. We believe it is unlikely that such an implementation exists because of the difficulty of 38.b/3

signalling other processors when the time reaches zero; in any case, depending on such an implementation is not portable.

D.14.3 Execution Time of Interrupt Handlers

1/3 {*AI05-0170-1*} {*AI05-0299-1*} This subclause describes a language-defined package to measure the execution time of interrupt handlers.

Static Semantics

2/3 {*AI05-0170-1*} The following language-defined library package exists:

3/5
```
{AI12-0241-1} {AI12-0302-1} with Ada.Interrupts;
package Ada.Execution_Time.Interrupts
   with Nonblocking, Global => in out synchronized is
   function Clock (Interrupt : Ada.Interrupts.Interrupt_Id)
        return CPU_Time;
   function Supported (Interrupt : Ada.Interrupts.Interrupt_Id)
        return Boolean;
end Ada.Execution_Time.Interrupts;
```

4/3 {*AI05-0170-1*} The execution time or CPU time of a given interrupt Interrupt is defined as the time spent by the system executing interrupt handlers identified by Interrupt, including the time spent executing run-time or system services on its behalf. The mechanism used to measure execution time is implementation defined. Time spent executing interrupt handlers is distinct from time spent executing any task.

4.a/3 **Discussion:** The implementation-defined mechanism here is the same as that covered by the Documentation Requirements of D.14, so we don't repeat that requirement here.

5/3 {*AI05-0170-1*} For each interrupt, the execution time value is initially set to zero.

Dynamic Semantics

6/3 {*AI05-0170-1*} The function Clock returns the current cumulative execution time of the interrupt identified by Interrupt. If Separate_Interrupt_Clocks_Supported is set to False the function raises Program_Error.

7/3 {*AI05-0170-1*} {*AI05-0264-1*} The function Supported returns True if the implementation is monitoring the execution time of the interrupt identified by Interrupt; otherwise, it returns False. For any Interrupt_Id Interrupt for which Supported(Interrupt) returns False, the function Clock(Interrupt) will return a value equal to Ada.Execution_Time.Time_Of(0).

Extensions to Ada 2005

7.a/3 {*AI05-0170-1*} The package Execution_Time.Interrupts is new.

D.15 Timing Events

1/5 {*AI95-00297-01*} {*AI05-0299-1*} {*AI12-0445-1*} This subclause describes a language-defined package to allow user-defined protected procedures to be executed at a specified time without the use of a task or a delay statement.

Static Semantics

2/2 {*AI95-00297-01*} The following language-defined library package exists:

3/5
```
{AI12-0241-1} {AI12-0302-1} package Ada.Real_Time.Timing_Events
  with Nonblocking, Global => in out synchronized is
```

```
{AI12-0241-1}   type Timing_Event is tagged limited private;                    4/5
  type Timing_Event_Handler
       is access protected procedure (Event : in out Timing_Event)
       with Nonblocking => False;

  procedure Set_Handler (Event   : in out Timing_Event;                         5/2
                         At_Time : in Time;
                         Handler : in Timing_Event_Handler);
  procedure Set_Handler (Event   : in out Timing_Event;
                         In_Time : in Time_Span;
                         Handler : in Timing_Event_Handler);
  function Current_Handler (Event : Timing_Event)
       return Timing_Event_Handler;
  procedure Cancel_Handler (Event     : in out Timing_Event;
                            Cancelled : out Boolean);

  function Time_Of_Event (Event : Timing_Event) return Time;                    6/2

private                                                                         7/2
   ... -- not specified by the language
end Ada.Real_Time.Timing_Events;
```

{*AI95-00297-01*} The type Timing_Event represents a time in the future when an event is to occur. The type Timing_Event needs finalization (see 7.6). 8/2

{*AI95-00297-01*} An object of type Timing_Event is said to be *set* if it is associated with a nonnull value of type Timing_Event_Handler and *cleared* otherwise. All Timing_Event objects are initially cleared. 9/2

{*AI95-00297-01*} The type Timing_Event_Handler identifies a protected procedure to be executed by the implementation when the timing event occurs. Such a protected procedure is called a *handler*. 10/2

Discussion: Type Timing_Event is tagged. This makes it possible to share a handler between several events. In simple cases, 'Access can be used to compare the parameter with a specific timing event object (this works because a tagged type is a by-reference type). In more complex cases, a type extension of type Timing_Event can be declared; a double type conversion can be used to access the extension data. For example: 10.a/2

```
type Toaster_Timing_Event is new Timing_Event with record                       10.b/2
   Slot : Natural;
end record;

...                                                                             10.c/2

protected body Toaster is                                                       10.d/2

   procedure Timer (Event : in out Timing_Event) is                             10.e/2
   begin
      Pop_Up_Toast (Toaster_Timing_Event(Timing_Event'Class(Event)).Slot);
   end Timer;

   ...                                                                          10.f/2
end Toaster;
```

The extra conversion to the class-wide type is necessary to make the conversions legal. While this usage is clearly ugly, we think that the need for this sort of usage will be rare, so we can live with it. It's certainly better than having no way to associate data with an event. 10.g/2

Dynamic Semantics

{*AI95-00297-01*} {*AI05-0264-1*} The procedures Set_Handler associate the handler Handler with the event Event: if Handler is **null**, the event is cleared; otherwise, it is set. The first procedure Set_Handler sets the execution time for the event to be At_Time. The second procedure Set_Handler sets the execution time for the event to be Real_Time.Clock + In_Time. 11/3

{*AI95-00297-01*} A call of a procedure Set_Handler for an event that is already set replaces the handler and the time of execution; if Handler is not **null**, the event remains set. 12/2

{*AI95-00297-01*} As soon as possible after the time set for the event, the handler is executed, passing the event as parameter. The handler is only executed if the timing event is in the set state at the time of execution. The initial action of the execution of the handler is to clear the event. 13/2

13.a/2 **Reason:** The second sentence of this paragraph is because of a potential race condition. The time might expire and yet before the handler is executed, some task could call Cancel_Handler (or equivalently call Set_Handler with a **null** parameter) and thus clear the handler.

14/2 {*AI95-00297-01*} If the Ceiling_Locking policy (see D.3) is in effect when a procedure Set_Handler is called, a check is made that the ceiling priority of Handler.**all** is Interrupt_Priority'Last. If the check fails, Program_Error is raised.

15/3 {*AI95-00297-01*} {*AI05-0094-1*} {*AI05-0264-1*} If a procedure Set_Handler is called with zero or negative In_Time or with At_Time indicating a time in the past, then the handler is executed as soon as possible after the completion of the call of Set_Handler.

15.a/3 **Ramification:** {*AI05-0094-1*} The handler will still be executed. Under no circumstances is a scheduled call of a handler lost.

15.b/3 **Discussion:** {*AI05-0094-1*} We say “as soon as possible” so that we do not deadlock if we are executing the handler when Set_Handler is called. In that case, the current invocation of the handler must complete before the new handler can start executing.

16/3 {*AI95-00297-01*} {*AI05-0264-1*} The function Current_Handler returns the handler associated with the event Event if that event is set; otherwise, it returns **null**.

17/3 {*AI95-00297-01*} {*AI05-0264-1*} The procedure Cancel_Handler clears the event if it is set. Cancelled is assigned True if the event was set prior to it being cleared; otherwise, it is assigned False.

18/3 {*AI95-00297-01*} {*AI05-0264-1*} The function Time_Of_Event returns the time of the event if the event is set; otherwise, it returns Real_Time.Time_First.

19/2 {*AI95-00297-01*} As part of the finalization of an object of type Timing_Event, the Timing_Event is cleared.

19.a/2 **Implementation Note:** This is the only finalization defined by the language that has a visible effect; but an implementation may have other finalization that it needs to perform. Implementations need to ensure that the event is cleared before anything else is finalized that would prevent a set event from being triggered.

20/2 {*AI95-00297-01*} If several timing events are set for the same time, they are executed in FIFO order of being set.

21/2 {*AI95-00297-01*} An exception propagated from a handler invoked by a timing event has no effect.

Implementation Requirements

22/2 {*AI95-00297-01*} For a given Timing_Event object, the implementation shall perform the operations declared in this package atomically with respect to any of these operations on the same Timing_Event object. The replacement of a handler by a call of Set_Handler shall be performed atomically with respect to the execution of the handler.

22.a/2 **Reason:** This prevents various race conditions. In particular it ensures that if an event occurs when Set_Handler is changing the handler then either the new or old handler is executed in response to the appropriate event. It is never possible for a new handler to be executed in response to an old event.

Metrics

23/2 {*AI95-00297-01*} The implementation shall document the following metric:

24/3 - {*AI05-0210-1*} An upper bound on the lateness of the execution of a handler. That is, the maximum time between the time specified for the event and when a handler is actually invoked assuming no other handler or task is executing during this interval.

24.a/2 **Documentation Requirement:** The metrics for timing events.

Implementation Advice

{*AI95-00297-01*} The protected handler procedure should be executed directly by the real-time clock interrupt mechanism. 25/2

Implementation Advice: For a timing event, the handler should be executed directly by the real-time clock interrupt mechanism. 25.a/2

NOTE 1 {*AI95-00297-01*} Since a call of Set_Handler is not a potentially blocking operation, it can be called from within a handler. 26/2

NOTE 2 {*AI95-00297-01*} A Timing_Event_Handler can be associated with several Timing_Event objects. 27/2

Extensions to Ada 95

{*AI95-00297-01*} The package Real_Time.Timing_Events is new. 27.a/2

Wording Changes from Ada 2005

{*AI05-0094-1*} **Correction:** Reworded to eliminate a deadlock condition if the event time is in the past and a handler is currently executing. This is technically an inconsistency, but only if a program is depending on deadlocking; since it is impossible to imagine how that could be useful, we have not documented this as an inconsistency. 27.b/3

{*AI05-0210-1*} **Correction:** Clarified the metric for lateness of a timing event to exclude interference from other handlers and tasks. This change might change the documentation of an implementation, but not the implementation itself, so there is no inconsistency. 27.c/3

D.16 Multiprocessor Implementation

{*AI05-0171-1*} {*AI05-0299-1*} This subclause allows implementations on multiprocessor platforms to be configured. 1/3

Static Semantics

{*AI05-0171-1*} The following language-defined library package exists: 2/3

```
{AI12-0241-1} {AI12-0302-1} package System.Multiprocessors                    3/5
   with Preelaborate, Nonblocking, Global => in out synchronized is

   type CPU_Range is range 0 .. implementation-defined;                       4/3
   Not_A_Specific_CPU : constant CPU_Range := 0;
   subtype CPU is CPU_Range range 1 .. CPU_Range'Last;
```

Implementation defined: The value of CPU_Range'Last in System.Multiprocessors. 4.a.1/3

```
   function Number_Of_CPUs return CPU;                                        5/3
end System.Multiprocessors;
```

{*AI05-0171-1*} A call of Number_Of_CPUs returns the number of processors available to the program. Within a given partition, each call on Number_Of_CPUs will return the same value. 6/3

{*AI05-0229-1*} {*AI12-0281-1*} For a task type (including the anonymous type of a single_task_declaration), protected type (including the anonymous type of a single_protected_declaration), or subprogram, the following language-defined representation aspect may be specified: 7/5

CPU The aspect CPU is an expression, which shall be of type System.Multiprocessors.CPU_Range. 8/3

Aspect Description for CPU: Processor on which a given task, or calling task for a protected operation, should run. 8.a/5

Legality Rules

{*AI05-0171-1*} {*AI05-0229-1*} If the CPU aspect is specified for a subprogram, the expression shall be static. 9/3

10/5 {*AI05-0229-1*} {*AI12-0281-1*} The CPU aspect shall not be specified on a task or protected interface type.

Dynamic Semantics

11/5 {*AI05-0171-1*} {*AI05-0229-1*} {*AI12-0081-1*} {*AI12-0281-1*} The expression specified for the CPU aspect of a task or protected type is evaluated each time an object of the corresponding type is created (see 9.1 and 9.4). The CPU value is then associated with the object.

12/3 {*AI05-0171-1*} {*AI05-0229-1*} The CPU aspect has no effect if it is specified for a subprogram other than the main subprogram; the CPU value is not associated with any task.

13/3 {*AI05-0171-1*} {*AI05-0229-1*} The CPU value is associated with the environment task if the CPU aspect is specified for the main subprogram. If the CPU aspect is not specified for the main subprogram it is implementation defined on which processor the environment task executes.

13.a.1/3 **Implementation defined:** The processor on which the environment task executes in the absence of a value for the aspect CPU.

14/5 {*AI05-0171-1*} {*AI05-0264-1*} {*AI12-0281-1*} For a task, the CPU value determines the processor on which the task will activate and execute; the task is said to be assigned to that processor. If the CPU value is Not_A_Specific_CPU, then the task is not assigned to a processor. A task without a CPU aspect specified will activate and execute on the same processor as its activating task if the activating task is assigned a processor. If the CPU value is not in the range of System.Multiprocessors.CPU_Range or is greater than Number_Of_CPUs the task is defined to have failed, and it becomes a completed task (see 9.2).

15/5 {*AI12-0281-1*} For a protected type, the CPU value determines the processor on which calling tasks will execute; the protected object is said to be assigned to that processor. If the CPU value is Not_A_Specific_CPU, then the protected object is not assigned to a processor. A call to a protected object that is assigned to a processor from a task that is not assigned a processor or is assigned a different processor raises Program_Error.

15.a/5 **Discussion:** {*AI12-0005-1*} When a protected object is assigned to a CPU, only tasks also assigned to that CPU can call it. In contrast, a protected object that is not assigned to a specific CPU can be called by any task on any processor (subject, of course, to visibility and ceiling priority restrictions). As noted below, when the tasks and protected object are necessarily on the same CPU, a simpler implementation can be used.

Implementation Advice

16/5 {*AI12-0281-1*} {*AI12-0323-1*} Starting a protected action on a protected object statically assigned to a processor should be implemented without busy-waiting.

16.a/5 **Reason:** Busy-waiting is a form of lock and can be a source of deadlock. Busy-waiting is typically needed for starting protected actions on multiprocessors, but if all tasks calling a protected object execute on the same CPU, this locking isn't needed and the source of deadlock and associated overhead can be eliminated.

16.b/5 **Implementation Advice:** Starting a protected action on a protected object statically assigned to a processor should not use busy-waiting.

Extensions to Ada 2005

16.c/3 {*AI05-0171-1*} {*AI05-0229-1*} The package System.Multiprocessors and the CPU aspect are new.

Extensions to Ada 2012

16.d/5 {*AI12-0281-1*} Aspect CPU can now be applied to protected types, in order to avoid the overhead and deadlock potential of multiprocessor execution.

Wording Changes from Ada 2012

16.e/4 {*AI12-0081-1*} **Corrigendum:** Clarified when the CPU aspect expression is evaluated.

D.16.1 Multiprocessor Dispatching Domains

{*AI05-0167-1*} {*AI05-0299-1*} This subclause allows implementations on multiprocessor platforms to be partitioned into distinct dispatching domains during program startup. 1/3

Static Semantics

{*AI05-0167-1*} The following language-defined library package exists: 2/3

```
{AI12-0241-1} {AI12-0302-1} with Ada.Real_Time;                                    3/5
with Ada.Task_Identification;
package System.Multiprocessors.Dispatching_Domains
   with Nonblocking, Global => in out synchronized is

   Dispatching_Domain_Error : exception;                                           4/3

   type Dispatching_Domain (<>) is limited private;                                5/3

   System_Dispatching_Domain : constant Dispatching_Domain;                        6/3

{AI12-0033-1}    function Create (First : CPU; Last : CPU_Range) return            7/4
Dispatching_Domain;

   function Get_First_CPU (Domain : Dispatching_Domain) return CPU;                8/3

{AI12-0033-1}    function Get_Last_CPU  (Domain : Dispatching_Domain) return       9/4
CPU_Range;

{AI12-0033-1}    type CPU_Set is array(CPU range <>) of Boolean;                   9.1/4

{AI12-0033-1}    function Create (Set : CPU_Set) return Dispatching_Domain;        9.2/4

{AI12-0033-1}    function Get_CPU_Set (Domain : Dispatching_Domain) return CPU_Set; 9.3/4

   function Get_Dispatching_Domain                                                 10/3
      (T   : Ada.Task_Identification.Task_Id :=
                Ada.Task_Identification.Current_Task)
           return Dispatching_Domain;

   procedure Assign_Task                                                           11/3
      (Domain : in out Dispatching_Domain;
       CPU    : in     CPU_Range := Not_A_Specific_CPU;
       T      : in     Ada.Task_Identification.Task_Id :=
                Ada.Task_Identification.Current_Task);

   procedure Set_CPU                                                               12/3
      (CPU : in CPU_Range;
       T   : in Ada.Task_Identification.Task_Id :=
                Ada.Task_Identification.Current_Task);

   function Get_CPU                                                                13/3
      (T   : Ada.Task_Identification.Task_Id :=
                Ada.Task_Identification.Current_Task)
           return CPU_Range;

   procedure Delay_Until_And_Set_CPU                                               14/3
      (Delay_Until_Time : in Ada.Real_Time.Time; CPU : in CPU_Range);

private                                                                            15/3
   ... -- not specified by the language
end System.Multiprocessors.Dispatching_Domains;
```

{*AI05-0167-1*} {*AI12-0082-1*} A *dispatching domain* represents a set of processors on which a task may execute. Each processor is contained within exactly one dispatching domain. An object of type Dispatching_Domain identifies a dispatching domain. System_Dispatching_Domain identifies a domain that contains the processor or processors on which the environment task executes. At program start-up all processors are contained within this domain. 16/4

{*AI05-0167-1*} For a task type (including the anonymous type of a single_task_declaration), the following language-defined representation aspect may be specified: 17/3

18/3 Dispatching_Domain

The value of aspect Dispatching_Domain is an expression, which shall be of type Dispatching_Domains.Dispatching_Domain. This aspect is the domain to which the task (or all objects of the task type) are assigned.

18.a/3 **Aspect Description for Dispatching_Domain:** Domain (group of processors) on which a given task should run.

Legality Rules

19/3 {*AI05-0167-1*} The Dispatching_Domain aspect shall not be specified for a task interface.

Dynamic Semantics

20/4 {*AI05-0167-1*} {*AI12-0033-1*} The expression specified for the Dispatching_Domain aspect of a task type is evaluated each time an object of the task type is created (see 9.1). If the identified dispatching domain is empty, then Dispatching_Domain_Error is raised; otherwise the newly created task is assigned to the domain identified by the value of the expression.

21/3 {*AI05-0167-1*} If a task is not explicitly assigned to any domain, it is assigned to that of the activating task. A task always executes on some CPU in its domain.

22/4 {*AI05-0167-1*} {*AI12-0082-1*} If both the dispatching domain and CPU are specified for a task, and the CPU value is not contained within the set of processors for the domain (and is not Not_A_Specific_CPU), the activation of the task is defined to have failed, and it becomes a completed task (see 9.2).

23/4 {*AI05-0167-1*} {*AI12-0033-1*} The function Create with First and Last parameters creates and returns a dispatching domain containing all the processors in the range First .. Last. The function Create with a Set parameter creates and returns a dispatching domain containing the processors for which Set(I) is True. These processors are removed from System_Dispatching_Domain. A call of Create will raise Dispatching_Domain_Error if any designated processor is not currently in System_Dispatching_Domain, or if the system cannot support a distinct domain over the processors identified, or if a processor has a task assigned to it, or if the allocation would leave System_Dispatching_Domain empty. A call of Create will raise Dispatching_Domain_Error if the calling task is not the environment task, or if Create is called after the call to the main subprogram.

24/4 {*AI05-0167-1*} {*AI12-0033-1*} The function Get_First_CPU returns the first CPU in Domain, or CPU'First if Domain is empty; Get_Last_CPU returns the last CPU in Domain, or CPU_Range'First if Domain is empty. The function Get_CPU_Set(D) returns an array whose low bound is Get_First_CPU(D), whose high bound is Get_Last_CPU(D), with True values in the Set corresponding to the CPUs that are in the given Domain.

25/4 {*AI05-0167-1*} {*AI12-0082-1*} The function Get_Dispatching_Domain returns the dispatching domain on which the task is assigned.

26/4 {*AI05-0167-1*} {*AI05-0278-1*} {*AI12-0033-1*} A call of the procedure Assign_Task assigns task T to the CPU within the dispatching domain Domain. Task T can now execute only on CPU, unless CPU designates Not_A_Specific_CPU in which case it can execute on any processor within Domain. The exception Dispatching_Domain_Error is propagated if Domain is empty, T is already assigned to a dispatching domain other than System_Dispatching_Domain, or if CPU is not one of the processors of Domain (and is not Not_A_Specific_CPU). A call of Assign_Task is a task dispatching point for task T unless T is inside of a protected action, in which case the effect on task T is delayed until its next task dispatching point. If T is the Current_Task the effect is immediate if T is not inside a protected action, otherwise the effect is as soon as practical. Assigning a task already assigned to System_Dispatching_Domain to that domain has no effect.

{*AI05-0167-1*} {*AI05-0278-1*} {*AI12-0082-1*} A call of procedure Set_CPU assigns task T to the CPU. Task T can now execute only on CPU, unless CPU designates Not_A_Specific_CPU, in which case it can execute on any processor within its dispatching domain. The exception Dispatching_Domain_Error is propagated if CPU is not one of the processors of the dispatching domain on which T is assigned (and is not Not_A_Specific_CPU). A call of Set_CPU is a task dispatching point for task T unless T is inside of a protected action, in which case the effect on task T is delayed until its next task dispatching point. If T is the Current_Task the effect is immediate if T is not inside a protected action, otherwise the effect is as soon as practical. 27/4

{*AI05-0167-1*} The function Get_CPU returns the processor assigned to task T, or Not_A_Specific_CPU if the task is not assigned to a processor. 28/3

{*AI05-0167-1*} {*AI12-0082-1*} A call of Delay_Until_And_Set_CPU delays the calling task for the designated time and then assigns the task to the specified processor when the delay expires. The exception Dispatching_Domain_Error is propagated if P is not one of the processors of the calling task's dispatching domain (and is not Not_A_Specific_CPU). 29/4

Implementation Requirements

{*AI05-0167-1*} The implementation shall perform the operations Assign_Task, Set_CPU, Get_CPU and Delay_Until_And_Set_CPU atomically with respect to any of these operations on the same dispatching_domain, processor or task. 30/3

{*AI12-0048-1*} Any task that belongs to the system dispatching domain can execute on any CPU within that domain, unless the assignment of the task has been specified. 30.1/4

Reason: This ensures that priorities and deadlines are respected within the system dispatching domain. There is no such guarantee between different domains. 30.a/4

We only need to talk about the system dispatching domain here, because Assign_Task and Set_CPU already have such wording for tasks that are assigned explicitly to a dispatching domain and specify Not_a_Specific_CPU. 30.b/4

Ramification: If no dispatching domains are created, all tasks can execute on all processors. (As always, implementation-defined dispatching policies may have other rules, so a partition that does not specify any language-defined dispatching policy may do anything at all and in particular does not need to follow this rule. 30.c/4

Discussion: A task can be assigned to a specific CPU by specifying the aspect CPU for a task, or by calling a dynamic operation like Set_CPU or Assign_Task. 30.d/4

Implementation Advice

{*AI05-0167-1*} Each dispatching domain should have separate and disjoint ready queues. 31/3

Implementation Advice: Each dispatching domain should have separate and disjoint ready queues. 31.a/3

To be honest: {*AI12-0048-1*} “Ready queue” here doesn't mean the conceptual "ready queue" as defined in D.2.1 (one per processor); this rule is talking about the ready queues used by the implementation. 31.b/4

Documentation Requirements

{*AI05-0167-1*} The implementation shall document the processor(s) on which the clock interrupt is handled and hence where delay queue and ready queue manipulations occur. For any Interrupt_Id whose handler can execute on more than one processor the implementation shall also document this set of processors. 32/3

Documentation Requirement: The processor(s) on which the clock interrupt is handled; the processors on which each Interrupt_Id can be handled. 32.a/3

Implementation Permissions

{*AI05-0167-1*} An implementation may limit the number of dispatching domains that can be created and raise Dispatching_Domain_Error if an attempt is made to exceed this number. 33/3

34/5 {*AI12-0119-1*} The implementation may defer the effect of a Set_CPU or an Assign_Task operation until the specified task leaves an ongoing parallel construct.

34.a/5 **Reason:** {*AI12-0119-1*} These operations can change the set of CPUs that a parallel operation is allowed to use. This could require the ability to move or suspend one or more threads to execute them on a different CPU. However, parallel constructs are primarily intended to improve performance of code, and the overhead needed to support such a rarely used operation could be substantial. Therefore, rather than requiring support we allow the implementation to wait to implement these operations until the parallel construct (and thus the extra threads) have completed.

Extensions to Ada 2005

34.b/3 {*AI05-0167-1*} {*AI05-0278-1*} The package System.Multiprocessors.Dispatching_Domains and the aspect Dispatching_Domains are new.

Inconsistencies With Ada 2012

34.c/4 {*AI12-0033-1*} **Corrigendum:** We now explicitly allow empty dispatching domains, as it would be difficult to avoid declaring them when a system is configured at runtime. Therefore, assigning a task to an empty domain now raises Dispatching_Domain_Error; creating such a domain should not raise Dispatching_Domain_Error. If an implementation does something different in these cases, and a program depends on that difference, the program could malfunction. This seems very unlikely (if no exception is ever raised, the task assigned to the empty domain could never run; if the exception is raised earlier, the program can't do anything useful).

Incompatibilities With Ada 2012

34.d/5 {*AI05-0033-1*} {*AI05-0005-1*} **Corrigendum:** The subtypes of the parameter or result of several routines were changed to support empty domains. These changes will cause rules requiring subtype conformance to fail on these routines (such as 'Access). We believe such uses are unlikely. In addition, type CPU_Set and function Get_CPU_Set, along with an overloaded Create are newly added to this package. As such, a use clause conflict is possible; see the introduction of Annex A for more on this topic.

Wording Changes from Ada 2012

34.e/4 {*AI12-0048-1*} **Corrigendum:** Added wording to clarify that all tasks can execute on all CPUs of the system dispatching domain by default.

34.f/4 {*AI12-0082-1*} **Corrigendum:** Added a definition to clarify that a "dispatching domain" is a concept which is identified by an object of type Dispatching_Domain; more than one object might identify the same dispatching domain (for instance, the result of function Get_Dispatching_Domain is a different object but identifies the same dispatching domain).

Annex E
(normative)
Distributed Systems

[This Annex defines facilities for supporting the implementation of distributed systems using multiple partitions working cooperatively as part of a single Ada program.] 1

Extensions to Ada 83

This Annex is new to Ada 95. 1.a

Post-Compilation Rules

A *distributed system* is an interconnection of one or more *processing nodes* (a system resource that has both computational and storage capabilities), and zero or more *storage nodes* (a system resource that has only storage capabilities, with the storage addressable by one or more processing nodes). 2

A *distributed program* comprises one or more partitions that execute independently (except when they communicate) in a distributed system. 3

The process of mapping the partitions of a program to the nodes in a distributed system is called *configuring the partitions of the program*. 4

Implementation Requirements

The implementation shall provide means for explicitly assigning library units to a partition and for the configuring and execution of a program consisting of multiple partitions on a distributed system; the means are implementation defined. 5

Implementation defined: The means for creating and executing distributed programs. 5.a

Implementation Permissions

An implementation may require that the set of processing nodes of a distributed system be homogeneous. 6

NOTE 1 {*AI12-0440-1*} The partitions comprising a program can be executed on differently configured distributed systems or on a nondistributed system without requiring recompilation. A distributed program can be partitioned differently from the same set of library units without recompilation. The resulting execution is semantically equivalent. 7/5

NOTE 2 A distributed program retains the same type safety as the equivalent single partition program. 8

E.1 Partitions

[The partitions of a distributed program are classified as either active or passive.] 1

Post-Compilation Rules

An *active partition* is a partition as defined in 10.2. A *passive partition* is a partition that has no thread of control of its own, whose library units are all preelaborated, and whose data and subprograms are accessible to one or more active partitions. 2

Discussion: In most situations, a passive partition does not execute, and does not have a “real” environment task. Any execution involved in its elaboration and initialization occurs before it comes into existence in a distributed program (like most preelaborated entities). Likewise, there is no concrete meaning to passive partition termination. 2.a

A passive partition shall include only library_items that either are declared pure or are shared passive (see 10.2.1 and E.2.1). 3

4 An active partition shall be configured on a processing node. A passive partition shall be configured either on a storage node or on a processing node.

5 The configuration of the partitions of a program onto a distributed system shall be consistent with the possibility for data references or calls between the partitions implied by their semantic dependences. Any reference to data or call of a subprogram across partitions is called a *remote access*.

5.a **Discussion:** For example, an active partition that includes a unit with a semantic dependence on the declaration of another RCI package of some other active partition has to be connected to that other partition by some sort of a message passing mechanism.

5.b A passive partition that is accessible to an active partition should have its storage addressable to the processor(s) of the active partition. The processor(s) should be able to read and write from/to that storage, as well as to perform "read-modify-write" operations (in order to support entry-less protected objects).

5.c/5 **Ramification:** {*AI12-0359-1*} A passive partition has no execution resources of its own, so while a call of a subprogram in a passive partition is a remote access to that subprogram, it is *not* a remote subprogram call (see E.4). The calling active partition executes the body of the subprogram of the passive partition.

Dynamic Semantics

6 A library_item is elaborated as part of the elaboration of each partition that includes it. If a normal library unit (see E.2) has state, then a separate copy of the state exists in each active partition that elaborates it. [The state evolves independently in each such partition.]

6.a **Ramification:** Normal library units cannot be included in passive partitions.

7 [An active partition *terminates* when its environment task terminates.] A partition becomes *inaccessible* if it terminates or if it is *aborted*. An active partition is aborted when its environment task is aborted. In addition, if a partition fails during its elaboration, it becomes inaccessible to other partitions. Other implementation-defined events can also result in a partition becoming inaccessible.

7.a **Implementation defined:** Any events that can result in a partition becoming inaccessible.

8/1 For a prefix D that denotes a library-level declaration, excepting a declaration of or within a declared-pure library unit, the following attribute is defined:

9 D'Partition_Id

Denotes a value of the type *universal_integer* that identifies the partition in which D was elaborated. If D denotes the declaration of a remote call interface library unit (see E.2.3) the given partition is the one where the body of D was elaborated.

Bounded (Run-Time) Errors

10/2 {*AI95-00226-01*} It is a bounded error for there to be cyclic elaboration dependences between the active partitions of a single distributed program. The possible effects, in each of the partitions involved, are deadlock during elaboration, or the raising of Communication_Error or Program_Error.

Implementation Permissions

11 An implementation may allow multiple active or passive partitions to be configured on a single processing node, and multiple passive partitions to be configured on a single storage node. In these cases, the scheduling policies, treatment of priorities, and management of shared resources between these partitions are implementation defined.

11.a **Implementation defined:** The scheduling policies, treatment of priorities, and management of shared resources between partitions in certain cases.

12 An implementation may allow separate copies of an active partition to be configured on different processing nodes, and to provide appropriate interactions between the copies to present a consistent state of the partition to other active partitions.

Ramification: The language does not specify the nature of these interactions, nor the actual level of consistency preserved. 12.a

{*AI12-0444-1*} In an implementation, the partitions of a distributed program may be loaded and elaborated at different times; they may be loaded and elaborated one at a time over an extended period of time. An implementation may provide facilities to abort and reload a partition during the execution of a distributed program. 13/5

An implementation may allow the state of some of the partitions of a distributed program to persist while other partitions of the program terminate and are later reinvoked. 14

NOTE 1 {*AI12-0417-1*} Library units are grouped into partitions after compile time, but before run time. At compile time, only the relevant library unit properties are identified using categorization aspects. 15/5

NOTE 2 The value returned by the Partition_Id attribute can be used as a parameter to implementation-provided subprograms in order to query information about the partition. 16

Wording Changes from Ada 95

{*AI95-00226-01*} Corrected wording so that a partition that has an elaboration problem will either deadlock or raise an exception. While an Ada 95 implementation could allow some partitions to continue to execute, they could be accessing unelaborated data, which is very bad (and erroneous in a practical sense). Therefore, this isn't listed as an inconsistency. 16.a/2

E.2 Categorization of Library Units

[Library units can be categorized according to the role they play in a distributed program. Certain restrictions are associated with each category to ensure that the semantics of a distributed program remain close to the semantics for a nondistributed program.] 1

{*AI05-0243-1*} {*AI12-0417-1*} A *categorization aspect* restricts the declarations, child units, or semantic dependences of the library unit to which it applies. A *categorized library unit* is a library unit that has a categorization aspect that is True. 2/3

{*AI05-0243-1*} {*AI12-0417-1*} The aspects Shared_Passive, Remote_Types, and Remote_Call_Interface are categorization aspects. In addition, for the purposes of this Annex, the aspect Pure (see 10.2.1) is considered a categorization aspect. 3/3

{*8652/0078*} {*AI95-00048-01*} {*AI05-0243-1*} [A library package or generic library package is called a *shared passive* library unit if the Shared_Passive aspect of the unit is True. A library package or generic library package is called a *remote types* library unit if the Remote_Types aspect of the unit is True. A library unit is called a *remote call interface* if the Remote_Call_Interface aspect of the unit is True.] A *normal library unit* is one for which no categorization aspect is True. 4/3

Proof: {*AI05-0243-1*} {*AI05-0299-1*} These terms (other than “normal library unit”) are really defined in the following subclauses. 4.a/3

Ramification: {*8652/0078*} {*AI95-00048-01*} A library subprogram can be a remote call interface, but it cannot be a remote types or shared passive library unit. 4.a.1/1

{*AI05-0206-1*} {*AI05-0243-1*} {*AI05-0269-1*} {*AI05-0299-1*} [The various categories of library units and the associated restrictions are described in this and the following subclauses. The categories are related hierarchically in that the library units of one category can depend semantically only on library units of that category or an earlier one in the hierarchy, except that the body of a remote types or remote call interface library unit is unrestricted, the declaration of a remote types or remote call interface library unit may depend on preelaborated normal library units that are mentioned only in private with clauses, and all categories can depend on limited views. 5/3

{*AI05-0243-1*} {*AI05-0269-1*} The overall hierarchy (including declared pure) is as follows, with a lower-numbered category being “earlier in the hierarchy” in the sense of the previous paragraph: 6/3

6.1/3 1. Declared Pure

6.2/3 2. Shared Passive

6.3/3 3. Remote Types

6.4/3 4. Remote Call Interface

6.5/3 5. Normal (no restrictions)

Paragraphs 7 through 11 were deleted.

12 Declared pure and shared passive library units are preelaborated. The declaration of a remote types or remote call interface library unit is required to be preelaborable.]

Paragraph 13 and 14 were deleted.

Wording Changes from Ada 95

14.a/2 {*8652/0078*} {*AI95-00048-01*} **Corrigendum:** Clarified that a library subprogram can be a remote call interface unit.

14.b/2 {*8652/0079*} {*AI95-00208-01*} **Corrigendum:** Removed the requirement that types be represented the same in all partitions, because it prevents the definition of heterogeneous distributed systems and goes much further than required.

Wording Changes from Ada 2005

14.c/3 {*AI05-0206-1*} {*AI05-0299-1*} We now allow private withs of preelaborated units in Remote Types and Remote Call Interface units; this is documented as an extension in the subclauses where this is defined normatively.

14.d/3 {*AI05-0243-1*} {*AI05-0299-1*} We have introduced categorization aspects; these are documented as extensions in the subclauses where they actually are defined.

Wording Changes from Ada 2012

14.e/5 {*AI12-0417-1*} Categorization pragmas are now obsolescent.

E.2.1 Shared Passive Library Units

1 [A shared passive library unit is used for managing global data shared between active partitions. The restrictions on shared passive library units prevent the data or tasks of one active partition from being accessible to another active partition through references implicit in objects declared in the shared passive library unit.]

Language Design Principles

1.a The restrictions governing a shared passive library unit are designed to ensure that objects and subprograms declared in the package can be used safely from multiple active partitions, even though the active partitions live in different address spaces, and have separate run-time systems.

Paragraphs 2 and 3 were moved to Annex J, "Obsolescent Features".

Legality Rules

4/5 {*AI05-0243-1*} {*AI12-0417-1*} When the library unit aspect (see 13.1.1) Shared_Passive of a library unit is True, the library unit is a *shared passive library unit*. The following restrictions apply to such a library unit:

4.a/3 **Aspect Description for Shared_Passive:** A given package is used to represent shared memory in a distributed system.

5 • [it shall be preelaborable (see 10.2.1);]

5.a **Ramification:** It cannot contain library-level declarations of protected objects with entries, nor of task objects. Task objects are disallowed because passive partitions don't have any threads of control of their own, nor any run-time system of their own. Protected objects with entries are disallowed because an entry queue contains references to calling tasks, and that would require in effect a pointer from a passive partition back to a task in some active partition.

- {*AI05-0243-1*} it shall depend semantically only upon declared pure or shared passive library_items; 6/3

  Reason: Shared passive packages cannot depend semantically upon remote types packages because the values of an access type declared in a remote types package refer to the local heap of the active partition including the remote types package. 6.a

  Ramification: {*AI05-0243-1*} We say library_item here, so that limited views are allowed; those are not library units, but they are library_item. 6.b/3

- {*8652/0080*} {*AI95-00003-01*} {*AI12-0038-1*} {*AI12-0320-1*} it shall not contain a library-level declaration: 7/5

  - {*AI12-0320-1*} of an access type that designates a class-wide type; 7.1/5

    Reason: These kinds of access types are disallowed because the object designated by an access value of such a type could contain an implicit reference back to the active partition on whose behalf the designated object was created. 7.a

  - {*AI12-0320-1*} of a type with a part that is of a task type; 7.2/5

  - {*AI12-0320-1*} of a type with a part that is of a protected type with entry_declarations; or 7.3/5

  - {*AI12-0038-1*} {*AI12-0320-1*} that contains a name that denotes a type declared within a declared-pure package, if that type has a part that is of an access type; for the purposes of this rule, the parts considered include those of the full views of any private types or private extensions. 7.4/5

    Reason: This rule breaks privacy by looking into the full views of private types. Avoiding privacy breakage here would have required disallowing the use in a shared passive package of any private type declared in a declared-pure package, which would have been severely incompatible. 7.b/4

Notwithstanding the definition of accessibility given in 3.10.2, the declaration of a library unit P1 is not accessible from within the declarative region of a shared passive library unit P2, unless the shared passive library unit P2 depends semantically on P1. 8

Discussion: We considered a more complex rule, but dropped it. This is the simplest rule that recognizes that a shared passive package may outlive some other library package, unless it depends semantically on that package. In a nondistributed program, all library packages are presumed to have the same lifetime. 8.a

{*AI12-0417-1*} Implementations may define additional aspects or pragmas that force two library packages to be in the same partition, or to have the same lifetime, which would allow this rule to be relaxed in the presence of such aspects or pragmas. 8.b/5

Static Semantics

A shared passive library unit is preelaborated. 9

Post-Compilation Rules

A shared passive library unit shall be assigned to at most one partition within a given program. 10

Notwithstanding the rule given in 10.2, a compilation unit in a given partition does not *need* (in the sense of 10.2) the shared passive library units on which it depends semantically to be included in that same partition; they will typically reside in separate passive partitions. 11

To be honest: {*AI12-0359-1*} This rule is necessary so that the shared data of a shared passive partition is not duplicated in each partition. It does not imply a particular implementation; in particular, code can be replicated in each active partition. 11.a/5

Implementation Note: {*AI12-0359-1*} One possible implementation for a shared passive partition is as a shared library that is mapped into the address space of of each active partition. In such case, both the code and the data of the passive partition would be mapped into the active partitions and directly called/accessed from each active partition. For instance, on Microsoft Windows a DLL has the correct semantics. 11.b/5

Alternatively, the shared data can be represented as a file or persistent memory, with the shared code being replicated in each active partition. Code replication is an as-if optimization; it should be impossible to tell where the code lives since no elaboration is necessary. 11.c/5

Wording Changes from Ada 95

11.d/2 {*8652/0080*} {*AI95-00003-01*} **Corrigendum:** Corrected the wording to allow access types in blocks in shared passive generic packages.

Extensions to Ada 2005

11.e/5 {*AI05-0243-1*} {*AI12-0417-1*} Shared_Passive is now a categorization aspect, so it can be specified by an aspect_specification

Incompatibilities With Ada 2012

11.f/4 {*AI12-0038-1*} **Corrigendum:** Uses of access types declared in declared-pure units are not allowed in library-level shared passive packages. These were allowed by Ada 2005 and Ada 2012, but it is unlikely that they work properly, as active partitions could disappear before the shared-passive partition. As such, the new errors are more likely to catch bugs than to cause them.

Wording Changes from Ada 2012

11.g/5 {*AI12-0417-1*} The pragma Shared_Passive is now obsolescent.

E.2.2 Remote Types Library Units

1 [A remote types library unit supports the definition of types intended for use in communication between active partitions.]

Language Design Principles

1.a The restrictions governing a remote types package are similar to those for a declared pure package. However, the restrictions are relaxed deliberately to allow such a package to contain declarations that violate the stateless property of pure packages, though it is presumed that any state-dependent properties are essentially invisible outside the package.

Paragraphs 2 and 3 were moved to Annex J, "Obsolescent Features".

Legality Rules

4/5 {*AI05-0243-1*} {*AI12-0417-1*} When the library unit aspect (see 13.1.1) Remote_Types of a library unit is True, the library unit is a *remote types library unit*. The following restrictions apply to such a library unit:

4.a/3 **Aspect Description for Remote_Types:** Types in a given package may be used in remote procedure calls.

5 • [it shall be preelaborable;]

6/3 • {*AI05-0206-1*} {*AI05-0243-1*} it shall depend semantically only on declared pure library_items, shared passive library units, other remote types library units, or preelaborated normal library units that are mentioned only in private with clauses;

6.a/3 **Ramification:** {*AI05-0243-1*} We say declared pure library_item here, so that (all) limited views are allowed; those are not library units, but they are declared pure library_items.

7 • it shall not contain the declaration of any variable within the visible part of the library unit;

7.a **Reason:** This is essentially a "methodological" restriction. A separate copy of a remote types package is included in each partition that references it, just like a normal package. Nevertheless, a remote types package is thought of as an "essentially pure" package for defining types to be used for interpartition communication, and it could be misleading to declare visible objects when no remote data access is actually being provided.

8/2 • {*AI95-00240-01*} {*AI95-00366-01*} the full view of each type declared in the visible part of the library unit that has any available stream attributes shall support external streaming (see 13.13.2).

8.a **Reason:** This is to prevent the use of the predefined Read and Write attributes of an access type as part of the Read and Write attributes of a visible type.

8.b/2 **Ramification:** {*AI95-00366-01*} Types that do not have available stream attributes are excluded from this rule; that means that attributes do not need to be specified for most limited types. It is only necessary to specify attributes for nonlimited types that have a part that is of any access type, and for extensions of limited types with available stream

attributes where the record_extension_part includes a subcomponent of an access type, where the access type does not have specified attributes.

{*8652/0082*} {*AI95-00164-01*} {*AI05-0060-1*} A named access type declared in the visible part of a remote types or remote call interface library unit is called a *remote access type*. Such a type shall be: 9/3

- {*8652/0082*} {*AI95-00164-01*} an access-to-subprogram type, or 9.1/1
- {*8652/0082*} {*AI95-00164-01*} {*AI05-0060-1*} a general access type that designates a class-wide limited private type, a class-wide limited interface type, or a class-wide private extension all of whose ancestors are either private extensions, limited interface types, or limited private types. 9.2/3

{*8652/0081*} {*AI95-00004-01*} A type that is derived from a remote access type is also a remote access type. 9.3/1

{*AI12-0283-1*} A remote access-to-subprogram type shall not be nonblocking (see 9.5). 9.4/5

Reason: All calls on remote subprograms are considered potentially blocking, so they cannot statically be allowed in nonblocking code. 9.a/5

Ramification: The type declaration of a remote type is illegal if the Nonblocking aspect is True, either implicitly by inheritance or by explicit specification. 9.b/5

The following restrictions apply to the use of a remote access-to-subprogram type: 10

- {*AI95-00431-01*} A value of a remote access-to-subprogram type shall be converted only to or from another (subtype-conformant) remote access-to-subprogram type; 11/2
- The prefix of an Access attribute_reference that yields a value of a remote access-to-subprogram type shall statically denote a (subtype-conformant) remote subprogram. 12

The following restrictions apply to the use of a remote access-to-class-wide type: 13

- {*8652/0083*} {*AI95-00047-01*} {*AI95-00240-01*} {*AI95-00366-01*} {*AI05-0060-1*} {*AI05-0101-1*} The primitive subprograms of the corresponding specific type shall only have access parameters if they are controlling formal parameters. The primitive functions of the corresponding specific type shall only have an access result if it is a controlling access result. Each noncontrolling formal parameter and noncontrolling result type shall support external streaming (see 13.13.2); 14/3
- {*AI05-0060-1*} {*AI05-0215-1*} {*AI05-0269-1*} The corresponding specific type shall not have a primitive procedure with the Synchronization aspect specified unless the synchronization_kind is Optional (see 9.5); 14.1/3
- A value of a remote access-to-class-wide type shall be explicitly converted only to another remote access-to-class-wide type; 15
- {*AI12-0034-1*} A value of a remote access-to-class-wide type shall be dereferenced (or implicitly converted to an anonymous access type) only as part of a dispatching call to a primitive operation of the designated type where the value designates a controlling operand of the call (see E.4, “Remote Subprogram Calls”); 16/4

  Ramification: {*AI12-0034-1*} Stream attributes of the designated type are not primitive operations of the designated type, and thus remote calls to them are prohibited by this rule. This is good, as the access parameter of a stream attribute does not have external streaming, and thus cannot be a parameter of a remote call. 16.a/4
- {*AI05-0101-1*} A controlling access result value for a primitive function with any controlling operands of the corresponding specific type shall either be explicitly converted to a remote access-to-class-wide type or be part of a dispatching call where the value designates a controlling operand of the call; 16.1/3
- {*AI95-00366-01*} {*AI12-0085-1*} The Storage_Pool attribute is not defined for a remote access-to-class-wide type; the expected type for an allocator shall not be a remote access-to-class-wide type. A remote access-to-class-wide type shall not be an actual parameter for a generic formal access type. The Storage_Size attribute of a remote access-to-class-wide type yields 0. The 17/4

Storage_Pool and Storage_Size aspects shall not be specified for a remote access-to-class-wide type.

17.a/3 **Reason:** {*AI05-0005-1*} All of these restrictions are because there is no storage pool associated with a remote access-to-class-wide type. The Storage_Size is defined to be 0 so that there is no conflict with the rules for pure units.

17.b/4 **Ramification:** {*AI12-0085-1*} The prohibition against specifying the Storage_Size aspect for an access-to-class-wide type applies to any method of doing that, including via either an aspect_specification or an attribute_definition_clause.

Erroneous Execution

17.1/4 {*AI12-0076-1*} Execution is erroneous if some operation (other than the initialization or finalization of the object) modifies the value of a constant object declared in the visible part of a remote types package.

17.c/4 **Discussion:** {*AI12-0005-1*} This could be accomplished via a self-referencing pointer or via squirrelling away a writable pointer to a controlled object.

18/5 NOTE 1 {*AI12-0442-1*} A remote types library unit is not necessarily pure, and the types it defines can include levels of indirection implemented by using access types. User-specified Read and Write attributes (see 13.13.2) provide for sending values of such a type between active partitions, with Write marshalling the representation, and Read unmarshalling any levels of indirection.

19/3 NOTE 2 {*AI05-0060-1*} The value of a remote access-to-class-wide limited interface can designate an object of a nonlimited type derived from the interface.

20/5 NOTE 3 {*AI05-0060-1*} {*AI12-0440-1*} A remote access type can designate a class-wide synchronized, protected, or task interface type.

20.a/3 **Proof:** Synchronized, protected, and task interfaces are all considered limited interfaces, see 3.9.4.

Incompatibilities With Ada 95

20.b/3 {*AI95-00240-01*} {*AI05-0248-1*} **Amendment Correction:** The wording was changed from "user-specified" to "available" read and write attributes. (This was then further changed, see below.) This means that an access type with the attributes specified in the private part would originally have been sufficient to allow the access type to be used in a remote type, but that is no longer allowed. Similarly, the attributes of a remote type that has access components have to be specified in the visible part. These changes were made so that the rules were consistent with the rules introduced for the Corrigendum for stream attributes; moreover, legality should not depend on the contents of the private part.

Extensions to Ada 95

20.c/3 {*AI95-00366-01*} {*AI05-0005-1*} Remote types that cannot be streamed (that is, have no available stream attributes) do not require the specification of stream attributes. This is necessary so that most extensions of Limited_Controlled do not need stream attributes defined (otherwise there would be a significant incompatibility, as Limited_Controlled would need stream attributes, and then all extensions of it also would need stream attributes).

Wording Changes from Ada 95

20.d/2 {*8652/0081*} {*AI95-00004-01*} **Corrigendum:** Added missing wording so that a type derived from a remote access type is also a remote access type.

20.e/2 {*8652/0083*} {*AI95-00047-01*} **Corrigendum:** Clarified that user-defined Read and Write attributes are required for the primitive subprograms corresponding to a remote access-to-class-wide type.

20.f/2 {*8652/0082*} {*AI95-00164-01*} **Corrigendum:** Added missing wording so that a remote access type can designate an appropriate private extension.

20.g/2 {*AI95-00366-01*} Changed the wording to use the newly defined term *type that supports external streaming*, so that various issues with access types in pure units and implicitly declared attributes for type extensions are properly handled.

20.h/2 {*AI95-00366-01*} Defined Storage_Size to be 0 for remote access-to-class-wide types, rather than having it undefined. This eliminates issues with pure units requiring a defined storage size.

20.i/2 {*AI95-00431-01*} Corrected the wording so that a value of a local access-to-subprogram type cannot be converted to a remote access-to-subprogram type, as intended (and required by the ACATS).

Incompatibilities With Ada 2005

20.j/5 {*AI05-0101-1*} {*AI12-0005-1*} {*AI12-0005-1*} **Correction:** Added rules about the returning of remote access-to-class-wide types; this had been missed in the past. While programs that returned unstreamable types from RCI functions

were legal, it is not clear what they could have done (as the results could not be marshalled). Similarly, RCI functions that return remote controlling access types could try to save those values, but it is unlikely that a compiler would know how to do that usefully. Thus, it seems unlikely that any real programs will be impacted by these changes.

Extensions to Ada 2005

{*AI05-0060-1*} **Correction:** Clarified that anonymous access types are never remote access types (and can be used in remote types units subject to the normal restrictions). Added wording to allow limited class-wide interfaces to be designated by remote access types. 20.k/3

{*AI05-0206-1*} Added wording to allow private withs of preelaborated normal units in the specification of a remote types unit. 20.l/3

{*AI05-0243-1*} {*AI12-0417-1*} Remote_Types is now a categorization aspect, so it can be specified by an aspect_specification 20.m/5

Wording Changes from Ada 2012

{*AI12-0034-1*} **Corrigendum:** Clarified that dispatching remote stream attribute calls are prohibited. We don't document this as an incompatibility, as the stream parameter cannot be marshalled for a remote call (it doesn't have external streaming), so it's impossible that any working program depends on this functionality. 20.n/4

{*AI12-0076-1*} **Corrigendum:** Explicitly stated that modifying a visible constant in a remote types package is erroneous. We don't document this as inconsistent as implementations certainly can still do whatever they were previously doing (no change is required); moreover, this case (and many more) were erroneous in Ada 2005 and before, so we're just restoring the previous semantics. 20.o/4

{*AI12-0085-1*} **Corrigendum:** Clarified that specifying the Storage_Pool or Storage_Size aspect for an access-to-class-wide type is not allowed. The intent is clear, and no implementation has ever allowed specifying the aspects (the attributes already cannot be specified), so we don't document this as an incompatibility. 20.p/4

{*AI12-0283-1*} Added a rule to ensure that potentially blocking remote calls are not considered nonblocking. 20.q/5

{*AI12-0417-1*} The pragma Remote_Types is now obsolescent. 20.r/5

E.2.3 Remote Call Interface Library Units

[A remote call interface library unit can be used as an interface for remote procedure calls (RPCs) (or remote function calls) between active partitions.] 1

Language Design Principles

The restrictions governing a remote call interface library unit are intended to ensure that the values of the actual parameters in a remote call can be meaningfully sent between two active partitions. 1.a

Paragraphs 2 through 6 were moved to Annex J, "Obsolescent Features".

Legality Rules

{*8652/0078*} {*AI95-00048-01*} {*AI05-0243-1*} {*AI12-0417-1*} When the library unit aspect (see 13.1.1) Remote_Call_Interface of a library unit is True, the library unit is a *remote call interface (RCI)*. A subprogram declared in the visible part of such a library unit, or declared by such a library unit, is called a *remote subprogram*. 7/5

Aspect Description for Remote_Call_Interface: Subprograms in a given package may be used in remote procedure calls. 7.a/3

{*AI05-0206-1*} {*AI05-0243-1*} The declaration of an RCI library unit shall be preelaborable (see 10.2.1), and shall depend semantically only upon declared pure library_items, shared passive library units, remote types library units, other remote call interface library units, or preelaborated normal library units that are mentioned only in private with clauses. 8/3

Ramification: {*AI05-0243-1*} We say declared pure library_item here, so that (all) limited views are allowed; those are not library units, but they are declared pure library_items. 8.a/3

{*8652/0078*} {*AI95-00048-01*} In addition, the following restrictions apply to an RCI library unit: 9/1

10/1 • {*8652/0078*} {*AI95-00048-01*} its visible part shall not contain the declaration of a variable;

10.a/1 **Reason:** {*8652/0078*} {*AI95-00048-01*} Remote call interface units do not provide remote data access. A shared passive package has to be used for that.

11/1 • {*8652/0078*} {*AI95-00048-01*} its visible part shall not contain the declaration of a limited type;

11.a/2 **Reason:** {*AI95-00240-01*} {*AI95-00366-01*} We disallow the declaration of task and protected types, since calling an entry or a protected subprogram implicitly passes an object of a limited type (the target task or protected object). We disallow other limited types since we require that such types have available Read and Write attributes, but we certainly don't want the Read and Write attributes themselves to involve remote calls (thereby defeating their purpose of marshalling the value for remote calls).

12/1 • {*8652/0078*} {*AI95-00048-01*} its visible part shall not contain a nested generic_declaration;

12.a **Reason:** This is disallowed because the body of the nested generic would presumably have access to data inside the body of the RCI package, and if instantiated in a different partition, remote data access might result, which is not supported.

13/3 • {*8652/0078*} {*AI95-00048-01*} {*AI05-0229-1*} it shall not be, nor shall its visible part contain, the declaration of a subprogram for which aspect Inline is True;

13.1/5 • {*AI12-0283-1*} it shall not be, nor shall its visible part contain, the declaration of a subprogram that is nonblocking (see 9.5);

13.a/5 **Reason:** All such subprograms are remote subprograms, and all calls on remote subprograms are considered potentially blocking, so they cannot statically be allowed in nonblocking code.

13.b/5 **Ramification:** The type declaration of a remote subprogram is illegal if the Nonblocking aspect is True, either implicitly by inheritance or by explicit specification.

14/3 • {*8652/0078*} {*AI95-00048-01*} {*AI95-00240-01*} {*AI95-00366-01*} {*AI05-0101-1*} it shall not be, nor shall its visible part contain, a subprogram (or access-to-subprogram) declaration whose profile has a parameter or result of a type that does not support external streaming (see 13.13.2);

14.a/3 **Ramification:** {*AI05-0101-1*} No anonymous access types support external streaming, so they are never allowed as parameters or results of RCI subprograms.

15 • any public child of the library unit shall be a remote call interface library unit.

15.a/5 **Reason:** {*AI12-0417-1*} No restrictions apply to the private part of an RCI package, and since a public child can “see” the private part of its parent, such a child must itself have a Remote_Call_Interface aspect, and be assigned to the same partition (see below).

15.b/5 **Discussion:** {*AI12-0417-1*} We considered making the public child of an RCI package implicitly RCI, but it seemed better to require an explicit aspect to avoid any confusion.

15.c Note that there is no need for a private child to be an RCI package, since it can only be seen from the body of its parent or its siblings, all of which are required to be in the same active partition.

15.1/5 {*AI12-0002-1*} Specification of a stream-oriented attribute is illegal in the specification of a remote call interface library unit. In addition to the places where Legality Rules normally apply (see 12.3), this rule applies also in the private part of an instance of a generic unit.

15.d/5 **Ramification:** This applies whether the stream-oriented attribute is specified with an attribute_definition_clause or with an aspect_specification.

15.e/5 **Reason:** Stream-oriented attributes are intended to be used locally (to marshall values), while subprograms in a remote call interface are intended to be used remotely. We resolve the conflict by banning the local use routines. The type should be included in a remote types package and imported into the remote call interface package.

16/5 {*AI05-0229-1*} {*AI12-0417-1*} All_Calls_Remote is a library unit aspect. If the All_Calls_Remote aspect of a library unit is True, the library unit shall be a remote call interface.

16.a/4 **Aspect Description for All_Calls_Remote:** All indirect or dispatching remote subprogram calls, and all direct remote subprogram calls, should use the Partition Communication Subsystem.

Post-Compilation Rules

A remote call interface library unit shall be assigned to at most one partition of a given program. A remote call interface library unit whose parent is also an RCI library unit shall be assigned only to the same partition as its parent. 17

Implementation Note: {*8652/0078*} {*AI95-00048-01*} The declaration of an RCI unit, with a calling-stub body, is automatically included in all active partitions with compilation units that depend on it. However the whole RCI library unit, including its (non-stub) body, will only be in one of the active partitions. 17.a/1

Notwithstanding the rule given in 10.2, a compilation unit in a given partition that semantically depends on the declaration of an RCI library unit, *needs* (in the sense of 10.2) only the declaration of the RCI library unit, not the body, to be included in that same partition. [Therefore, the body of an RCI library unit is included only in the partition to which the RCI library unit is explicitly assigned.] 18

Implementation Requirements

{*8652/0078*} {*AI95-00048-01*} {*AI05-0229-1*} {*AI12-0031-1*} If aspect All_Calls_Remote is True for a given RCI library unit, then the implementation shall route any of the following calls through the Partition Communication Subsystem (PCS); see E.5: 19/4

- {*AI12-0031-1*} A direct call to a subprogram of the RCI unit from outside the declarative region of the unit; 19.1/4
- {*AI12-0031-1*} An indirect call through a remote access-to-subprogram value that designates a subprogram of the RCI unit; 19.2/4
- {*AI12-0031-1*} A dispatching call with a controlling operand designated by a remote access-to-class-wide value whose tag identifies a type declared in the RCI unit. 19.3/4

Discussion: {*8652/0078*} {*AI95-00048-01*} {*AI05-0229-1*} {*AI12-0031-1*} {*AI12-0005-1*} When this aspect is False (or not used), it is presumed that most implementations will not make calls through the PCS if the call originates in the same partition as that of the RCI unit. When this aspect is True, all indirect or dispatching remote subprogram calls to the RCI unit, and all direct calls from outside the subsystem rooted at the RCI unit, are treated like calls from outside the partition, ensuring that the PCS is involved in all such calls (for debugging, redundancy, etc.). 19.a/5

Reason: {*AI12-0031-1*} {*AI12-0005-1*} There is no point in forcing local direct calls (including calls from children) to go through the PCS, since on the target system these calls are always local, and all the units are in the same active partition. 19.b/5

Implementation Permissions

{*AI05-0243-1*} {*AI12-0417-1*} {*AI12-0444-1*} An implementation may omit support for the Remote_Call_Interface aspect or the All_Calls_Remote aspect. [Explicit message-based communication between active partitions can be supported as an alternative to RPC.] 20/5

Ramification: {*AI12-0417-1*} Of course, it is pointless to support the All_Calls_Remote aspect if the Remote_Call_Interface aspect (or some approximate equivalent) is not supported. 20.a/5

Incompatibilities With Ada 95

{*AI95-00240-01*} {*AI05-0248-1*} **Amendment Correction:** The wording was changed from “user-specified” to “available” read and write attributes. (This was then further changed, see below.) This means that a type with the attributes specified in the private part would originally have been allowed as a formal parameter of an RCI subprogram, but that is no longer allowed. This change was made so that the rules were consistent with the rules introduced for the Corrigendum for stream attributes; moreover, legality should not depend on the contents of the private part. 20.b/3

Wording Changes from Ada 95

{*8652/0078*} {*AI95-00048-01*} **Corrigendum:** Changed the wording to allow a library subprogram to be a remote call interface unit. 20.c/2

20.d/2 {*AI95-00366-01*} Changed the wording to use the newly defined term *type that supports external streaming*, so that various issues with access types in pure units and implicitly declared attributes for type extensions are properly handled.

Incompatibilities With Ada 2005

20.e/3 {*AI05-0101-1*} **Correction:** Added a rule to ensure that function results are streamable; this was missing in previous versions of Ada. While programs that returned unstreamable types from RCI functions were legal, it is not clear what they could have done (as the results could not be marshalled). Thus, it seems unlikely that any real programs will be impacted by this change.

Extensions to Ada 2005

20.f/3 {*AI05-0206-1*} Added wording to allow private withs of preelaborated normal units in the specification of a remote call interface unit.

20.g/5 {*AI05-0229-1*} {*AI12-0417-1*} All_Calls_Remote is now a representation aspect, so it can be specified by an aspect_specification

20.h/5 {*AI05-0243-1*} {*AI12-0417-1*} Remote_Call_Interface is now a categorization aspect, so it can be specified by an aspect_specification

Inconsistencies With Ada 2012

20.i/4 {*AI12-0031-1*} **Corrigendum:** Redefined when indirect and dispatching remote calls have to be remote for a unit for which the aspect All_Calls_Remote is True. With the new rules, a local target called indirectly or via dispatching will be routed through the PCS, while that was not necessarily true in earlier Ada. If a program depended on local targets not being routed through the PCS even when All_Calls_Remote is used, then it might behave differently or fail in Ada 2012 as corrected with this refinement. This is highly unlikely as the PCS is going to be able to communicate with any partition of the program, including the local partition.

Incompatibilities With Ada 2012

20.j/5 {*AI12-0002-1*} **Correction:** Specifying user-defined stream-oriented attributes is now illegal in an RCI unit. If a program contains such attributes, they and their associated type will need to be moved to a remote types package (which is then imported into the RCI unit).

Wording Changes from Ada 2012

20.k/5 {*AI12-0283-1*} Added a rule to ensure that potentially blocking remote calls are not considered nonblocking.

20.l/5 {*AI12-0417-1*} The pragmas Remote_Call_Interface and All_Calls_Remote are now obsolescent.

E.3 Consistency of a Distributed System

1/3 {*AI05-0299-1*} [This subclause defines attributes and rules associated with verifying the consistency of a distributed program.]

Language Design Principles

1.a/3 {*AI05-0248-1*} The rules guarantee that remote call interface and shared passive library units are consistent among all partitions prior to the execution of a distributed program, so that the semantics of the distributed program are well defined.

Static Semantics

2/1 For a prefix P that statically denotes a program unit, the following attributes are defined:

3 P'Version Yields a value of the predefined type String that identifies the version of the compilation unit that contains the declaration of the program unit.

4 P'Body_Version
Yields a value of the predefined type String that identifies the version of the compilation unit that contains the body (but not any subunits) of the program unit.

5/1 {*8652/0084*} {*AI95-00104-01*} The *version* of a compilation unit changes whenever the compilation unit changes in a semantically significant way. This document does not define the exact meaning of

"semantically significant". It is unspecified whether there are other events (such as recompilation) that result in the version of a compilation unit changing.

This paragraph was deleted. 5.a/1

{*8652/0084*} {*AI95-00104-01*} If P is not a library unit, and P has no completion, then P'Body_Version returns the Body_Version of the innermost program unit enclosing the declaration of P. If P is a library unit, and P has no completion, then P'Body_Version returns a value that is different from Body_Version of any version of P that has a completion. 5.1/1

Bounded (Run-Time) Errors

In a distributed program, a library unit is *consistent* if the same version of its declaration is used throughout. It is a bounded error to elaborate a partition of a distributed program that contains a compilation unit that depends on a different version of the declaration of a shared passive or RCI library unit than that included in the partition to which the shared passive or RCI library unit was assigned. As a result of this error, Program_Error can be raised in one or both partitions during elaboration; in any case, the partitions become inaccessible to one another. 6

Ramification: Because a version changes if anything on which it depends undergoes a version change, requiring consistency for shared passive and remote call interface library units is sufficient to ensure consistency for the declared pure and remote types library units that define the types used for the objects and parameters through which interpartition communication takes place. 6.a

Note that we do not require matching Body_Versions; it is irrelevant for shared passive and remote call interface packages, since only one copy of their body exists in a distributed program (in the absence of implicit replication), and we allow the bodies to differ for declared pure and remote types packages from partition to partition, presuming that the differences are due to required error corrections that took place during the execution of a long-running distributed program. The Body_Version attribute provides a means for performing stricter consistency checks. 6.b

Wording Changes from Ada 95

{*8652/0084*} {*AI95-00104-01*} **Corrigendum:** Clarified the meaning of 'Version and 'Body_Version. 6.c/2

E.4 Remote Subprogram Calls

{*AI12-0359-1*} A *remote subprogram call* is a subprogram call that invokes the execution of a subprogram in another (active) partition. The partition that originates the remote subprogram call is the *calling partition*, and the partition that executes the corresponding subprogram body is the *called partition*. Some remote procedure calls are allowed to return prior to the completion of subprogram execution. These are called *asynchronous remote procedure calls*. 1/5

Discussion: {*AI12-0359-1*} Remote subprogram calls are always between active partitions; a passive partition has no execution resources of its own and thus cannot execute anything, while a remote subprogram call is always executed by the called partition. 1.a/5

There are three different ways of performing a remote subprogram call: 2

- As a direct call on a (remote) subprogram explicitly declared in a remote call interface; 3
- As an indirect call through a value of a remote access-to-subprogram type; 4
- As a dispatching call with a controlling operand designated by a value of a remote access-to-class-wide type. 5

The first way of calling corresponds to a *static* binding between the calling and the called partition. The latter two ways correspond to a *dynamic* binding between the calling and the called partition. 6

{*AI05-0101-1*} Remote types library units (see E.2.2) and remote call interface library units (see E.2.3) define the remote subprograms or remote access types used for remote subprogram calls. 7/3

Language Design Principles

7.a Remote subprogram calls are standardized since the RPC paradigm is widely-used, and establishing an interface to it in the annex will increase the portability and reusability of distributed programs.

Legality Rules

8 In a dispatching call with two or more controlling operands, if one controlling operand is designated by a value of a remote access-to-class-wide type, then all shall be.

8.1/5 {*AI12-0283-1*} A nonblocking program unit shall not contain, other than within nested units with Nonblocking specified as statically False, a dispatching call with a controlling operand designated by a value of a remote access-to-class-wide type.

8.a/5 **Reason:** Such a dispatching call is a potentially blocking call (see below) even if the called subprogram is nonblocking, so we must not assert that no blocking is possible.

8.b/5 **Ramification:** The calls is illegal if the Nonblocking aspect of the containing unit is True, either implicitly by inheritance or by explicit specification.

Dynamic Semantics

9 For the execution of a remote subprogram call, subprogram parameters (and later the results, if any) are passed using a stream-oriented representation (see 13.13.1) [which is suitable for transmission between partitions]. This action is called *marshalling*. *Unmarshalling* is the reverse action of reconstructing the parameters or results from the stream-oriented representation. [Marshalling is performed initially as part of the remote subprogram call in the calling partition; unmarshalling is done in the called partition. After the remote subprogram completes, marshalling is performed in the called partition, and finally unmarshalling is done in the calling partition.]

10 A *calling stub* is the sequence of code that replaces the subprogram body of a remotely called subprogram in the calling partition. A *receiving stub* is the sequence of code (the “wrapper”) that receives a remote subprogram call on the called partition and invokes the appropriate subprogram body.

10.a **Discussion:** The use of the term *stub* in this annex should not be confused with body_stub as defined in 10.1.3. The term *stub* is used here because it is a commonly understood term when talking about the RPC paradigm.

11 Remote subprogram calls are executed at most once, that is, if the subprogram call returns normally, then the called subprogram's body was executed exactly once.

12 The task executing a remote subprogram call blocks until the subprogram in the called partition returns, unless the call is asynchronous. For an asynchronous remote procedure call, the calling task can become ready before the procedure in the called partition returns.

13 If a construct containing a remote call is aborted, the remote subprogram call is *cancelled*. Whether the execution of the remote subprogram is immediately aborted as a result of the cancellation is implementation defined.

13.a **Implementation defined:** Whether the execution of the remote subprogram is immediately aborted as a result of cancellation.

14 If a remote subprogram call is received by a called partition before the partition has completed its elaboration, the call is kept pending until the called partition completes its elaboration (unless the call is cancelled by the calling partition prior to that).

15 If an exception is propagated by a remotely called subprogram, and the call is not an asynchronous call, the corresponding exception is reraised at the point of the remote subprogram call. For an asynchronous call, if the remote procedure call returns prior to the completion of the remotely called subprogram, any exception is lost.

The exception Communication_Error (see E.5) is raised if a remote call cannot be completed due to difficulties in communicating with the called partition. 16

{*AI12-0183-1*} All forms of remote subprogram calls are potentially blocking operations (see 9.5). 17/5

Reason: Asynchronous remote procedure calls are potentially blocking since the implementation may require waiting for the availability of shared resources to initiate the remote call. 17.a

{*8652/0085*} {*AI95-00215-01*} In a remote subprogram call with a formal parameter of a class-wide type, a check is made that the tag of the actual parameter identifies a tagged type declared in a declared-pure or shared passive library unit, or in the visible part of a remote types or remote call interface library unit. Program_Error is raised if this check fails. In a remote function call which returns a class-wide type, the same check is made on the function result. 18/1

Discussion: {*8652/0085*} {*AI95-00215-01*} This check makes certain that the specific type passed or returned in an RPC satisfies the rules for a "communicable" type. Normally this is guaranteed by the compile-time restrictions on remote call interfaces. However, with class-wide types, it is possible to pass an object whose tag identifies a type declared outside the "safe" packages. 18.a/1

This is considered an accessibility_check since only the types declared in "safe" packages are considered truly "global" (cross-partition). Other types are local to a single partition. This is analogous to the "accessibility" of global vs. local declarations in a single-partition program. 18.b

This rule replaces a rule from an early version of Ada 9X which was given in the subclause on Remote Types Library Units (now E.2.2, “Remote Types Library Units”). That rule tried to prevent "bad" types from being sent by arranging for their tags to mismatch between partitions. However, that interfered with other uses of tags. The new rule allows tags to agree in all partitions, even for those types which are not "safe" to pass in an RPC. 18.c

In a dispatching call with two or more controlling operands that are designated by values of a remote access-to-class-wide type, a check is made [(in addition to the normal Tag_Check — see 11.5)] that all the remote access-to-class-wide values originated from Access attribute_references that were evaluated by tasks of the same active partition. Constraint_Error is raised if this check fails. 19

Implementation Note: When a remote access-to-class-wide value is created by an Access attribute_reference, the identity of the active partition that evaluated the attribute_reference should be recorded in the representation of the remote access value. 19.a

Implementation Requirements

The implementation of remote subprogram calls shall conform to the PCS interface as defined by the specification of the language-defined package System.RPC (see E.5). The calling stub shall use the Do_RPC procedure unless the remote procedure call is asynchronous in which case Do_APC shall be used. On the receiving side, the corresponding receiving stub shall be invoked by the RPC-receiver. 20

Implementation Note: One possible implementation model is as follows: 20.a

The code for calls to subprograms declared in an RCI package is generated normally, that is, the call-site is the same as for a local subprogram call. The code for the remotely callable subprogram bodies is also generated normally. Subprogram's prologue and epilogue are the same as for a local call. 20.b

When compiling the specification of an RCI package, the compiler generates calling stubs for each visible subprogram. Similarly, when compiling the body of an RCI package, the compiler generates receiving stubs for each visible subprogram together with the appropriate tables to allow the RPC-receiver to locate the correct receiving stub. 20.c

For the statically bound remote calls, the identity of the remote partition is statically determined (it is resolved at configuration/link time). 20.d

The calling stub operates as follows: 20.e

- It allocates (or reuses) a stream of Params_Stream_Type of Initial_Size, and initializes it by repeatedly calling Write operations, first to identify which remote subprogram in the receiving partition is being called, and then to pass the incoming value of each of the **in** and **in out** parameters of the call. 20.f
- {*AI05-0229-1*} It allocates (or reuses) a stream for the Result, unless an aspect Asynchronous is specified as True for the procedure. 20.g/3
- {*AI05-0229-1*} It calls Do_RPC unless an aspect Asynchronous is specified as True for the procedure in which case it calls Do_APC. An access value designating the message stream allocated and initialized above 20.h/3

is passed as the Params parameter. An access value designating the Result stream is passed as the Result parameter.

20.i/3 • {*AI05-0229-1*} If the aspect Asynchronous is not specified for the procedure, Do_RPC blocks until a reply message arrives, and then returns to the calling stub. The stub returns after extracting from the Result stream, using Read operations, the **in out** and **out** parameters or the function result. If the reply message indicates that the execution of the remote subprogram propagated an exception, the exception is propagated from Do_RPC to the calling stub, and thence to the point of the original remote subprogram call. If Do_RPC detects that communication with the remote partition has failed, it propagates Communication_Error.

20.j On the receiving side, the RPC-receiver procedure operates as follows:

20.k • It is called from the PCS when a remote-subprogram-call message is received. The call originates in some remote call receiver task executed and managed in the context of the PCS.

20.l • It extracts information from the stream to identify the appropriate receiving stub.

20.m • The receiving stub extracts the **in** and **in out** parameters using Read from the stream designated by the Params parameter.

20.n • The receiving stub calls the actual subprogram body and, upon completion of the subprogram, uses Write to insert the results into the stream pointed to by the Result parameter. The receiving stub returns to the RPC-receiver procedure which in turn returns to the PCS. If the actual subprogram body propagates an exception, it is propagated by the RPC-receiver to the PCS, which handles the exception, and indicates in the reply message that the execution of the subprogram body propagated an exception. The exception occurrence can be represented in the reply message using the Write attribute of Ada.Exceptions.Exception_Occurrence.

20.o For remote access-to-subprogram types:

20.p A value of a remote access-to-subprogram type can be represented by the following components: a reference to the remote partition, an index to the package containing the remote subprogram, and an index to the subprogram within the package. The values of these components are determined at run time when the remote access value is created. These three components serve the same purpose when calling Do_APC/RPC, as in the statically bound remote calls; the only difference is that they are evaluated dynamically.

20.q For remote access-to-class-wide types:

20.r For each remote access-to-class-wide type, a calling stub is generated for each dispatching operation of the designated type. In addition, receiving stubs are generated to perform the remote dispatching operations in the called partition. The appropriate subprogram_body is determined as for a local dispatching call once the receiving stub has been reached.

20.s A value of a remote access-to-class-wide type can be represented with the following components: a reference to the remote partition, an index to a table (created one per each such access type) containing addresses of all the dispatching operations of the designated type, and an access value designating the actual remote object.

20.t Alternatively, a remote access-to-class-wide value can be represented as a normal access value, pointing to a "stub" object which in turn contains the information mentioned above. A call on any dispatching operation of such a stub object does the remote call, if necessary, using the information in the stub object to locate the target partition, etc. This approach has the advantage that less special-casing is required in the compiler. All access values can remain just a simple address.

20.u For a call to Do_RPC or Do_APC: The partition ID of all controlling operands are checked for equality (a Constraint_Error is raised if this check fails). The partition ID value is used for the Partition parameter. An index into the *tagged-type-descriptor* is created. This index points to the receiving stub of the class-wide operation. This index and the index to the table (described above) are written to the stream. Then, the actual parameters are marshalled into the message stream. For a controlling operand, only the access value designating the remote object is required (the other two components are already present in the other parameters).

20.v On the called partition (after the RPC-receiver has transferred control to the appropriate receiving stub) the parameters are first unmarshalled. Then, the tags of the controlling operands (obtained by dereferencing the pointer to the object) are checked for equality. If the check fails Constraint_Error is raised and propagated back to the calling partition, unless it is a result of an asynchronous call. Finally, a dispatching call to the specific subprogram (based on the controlling object's tag) is made. Note that since this subprogram is not in an RCI package, no specific stub is generated for it, it is called normally from the *dispatching stub*.

20.1/1 {*8652/0086*} {*AI95-00159-01*} With respect to shared variables in shared passive library units, the execution of the corresponding subprogram body of a synchronous remote procedure call is considered to be part of the execution of the calling task. The execution of the corresponding subprogram body of an asynchronous remote procedure call proceeds in parallel with the calling task and does not signal the next action of the calling task (see 9.10).

NOTE 1 A given active partition can both make and receive remote subprogram calls. Thus, an active partition can act as both a client and a server. 21

NOTE 2 If a given exception is propagated by a remote subprogram call, but the exception does not exist in the calling partition, the exception can be handled by an **others** choice or be propagated to and handled by a third partition. 22

Discussion: This situation can happen in a case of dynamically nested remote subprogram calls, where an intermediate call executes in a partition that does not include the library unit that defines the exception. 22.a

Wording Changes from Ada 95

{*8652/0086*} {*AI95-00159-01*} **Corrigendum:** Added rules so that tasks can safely access shared passive objects. 22.b/2

{*8652/0085*} {*AI95-00215-01*} **Corrigendum:** Clarified that the check on class-wide types also applies to values returned from remote subprogram call functions. 22.c/2

Wording Changes from Ada 2005

{*AI05-0101-1*} **Correction:** Corrected the text to note that remote access types can be defined in remote types units. 22.d/3

Wording Changes from Ada 2012

{*AI12-0283-1*} Added a rule to ensure that potentially blocking remote calls are not considered nonblocking. 22.e/5

{*AI12-0359-1*} Clarified that remote subprogram calls are always to active partitions. 22.f/5

E.4.1 Asynchronous Remote Calls

{*AI05-0229-1*} [This subclause introduces the aspect Asynchronous which can be specified to allow a remote subprogram call to return prior to completion of the execution of the corresponding remote subprogram body.] 1/3

Paragraphs 2 through 7 were deleted.

Static Semantics

{*AI05-0229-1*} For a remote procedure, the following language-defined representation aspect may be specified: 8/3

Asynchronous 8.1/3

The type of aspect Asynchronous is Boolean. If directly specified, the aspect_definition shall be a static expression. If not specified, the aspect is False.

Aspect Description for Asynchronous: Remote procedure calls are asynchronous; the caller continues without waiting for the call to return. 8.a/3

{*AI05-0229-1*} For a remote access type, the following language-defined representation aspect may be specified: 8.2/3

Asynchronous 8.3/3

The type of aspect Asynchronous is Boolean. If directly specified, the aspect_definition shall be a static expression. If not specified (including by inheritance), the aspect is False.

Legality Rules

{*AI05-0229-1*} If aspect Asynchronous is specified for a remote procedure, the formal parameters of the procedure shall all be of mode **in**. 8.4/3

{*AI05-0229-1*} If aspect Asynchronous is specified for a remote access type, the type shall be a remote access-to-class-wide type, or the type shall be a remote access-to-procedure type with the formal parameters of the designated profile of the type all of mode **in**. 8.5/3

Dynamic Semantics

9/3 {*AI05-0229-1*} A remote call is *asynchronous* if it is a call to a procedure, or a call through a value of an access-to-procedure type, for which aspect Asynchronous is True. In addition, if aspect Asynchronous is True for a remote access-to-class-wide type, then a dispatching call on a procedure with a controlling operand designated by a value of the type is asynchronous if the formal parameters of the procedure are all of mode **in**.

Implementation Requirements

10 Asynchronous remote procedure calls shall be implemented such that the corresponding body executes at most once as a result of the call.

10.a **To be honest:** It is not clear that this rule can be tested or even defined formally.

Extensions to Ada 2005

10.b/3 {*AI05-0229-1*} Aspect Asynchronous is new; pragma Asynchronous is now obsolescent.

E.4.2 Example of Use of a Remote Access-to-Class-Wide Type

Examples

1 *Example of using a remote access-to-class-wide type to achieve dynamic binding across active partitions:*

2/5
```
{AI12-0414-1} package Tapes
   with Pure is
   type Tape is abstract tagged limited private;
   -- Primitive dispatching operations where
   -- Tape is controlling operand
   procedure Copy (From, To : access Tape;
                   Num_Recs : in Natural) is abstract;
   procedure Rewind (T : access Tape) is abstract;
   -- More operations
private
   type Tape is ...
end Tapes;
```

3/5
```
{AI12-0414-1} with Tapes;
package Name_Server
   with Remote_Call_Interface is
   -- Dynamic binding to remote operations is achieved
   -- using the access-to-limited-class-wide type Tape_Ptr
   type Tape_Ptr is access all Tapes.Tape'Class;
   -- The following statically bound remote operations
   -- allow for a name-server capability in this example
   function  Find     (Name : String) return Tape_Ptr;
   procedure Register (Name : in String; T : in Tape_Ptr);
   procedure Remove   (T : in Tape_Ptr);
   -- More operations
end Name_Server;
```

4
```
package Tape_Driver is
  -- Declarations are not shown, they are irrelevant here
end Tape_Driver;
```

```
{AI12-0347-1} with Tapes, Name_Server;                                    5/5
package body Tape_Driver is
   type New_Tape is new Tapes.Tape with ...
   overriding
   procedure Rewind (T : access New_Tape);
   overriding
   procedure Copy
     (From, To : access New_Tape; Num_Recs: in Natural) is
   begin
      . . .
   end Copy;
   procedure Rewind (T : access New_Tape) is
   begin
      . . .
   end Rewind;
   -- Objects remotely accessible through use
   -- of Name_Server operations
   Tape1, Tape2 : aliased New_Tape;
begin
   Name_Server.Register ("NINE-TRACK",  Tape1'Access);
   Name_Server.Register ("SEVEN-TRACK", Tape2'Access);
end Tape_Driver;
```

```
with Tapes, Name_Server;                                                  6
-- Tape_Driver is not needed and thus not mentioned in the with_clause
procedure Tape_Client is
   T1, T2 : Name_Server.Tape_Ptr;
begin
   T1 := Name_Server.Find ("NINE-TRACK");
   T2 := Name_Server.Find ("SEVEN-TRACK");
   Tapes.Rewind (T1);
   Tapes.Rewind (T2);
   Tapes.Copy (T1, T2, 3);
end Tape_Client;
```

{*AI12-0452-1*} *Discussion of the example*: 7/5

Discussion: The example does not show the case where tapes are removed from or added to the system. In the former case, an appropriate exception needs to be defined to instruct the client to use another tape. In the latter, the Name_Server should have a query function visible to the clients to inform them about the availability of the tapes in the system. 7.a

This paragraph was deleted. 8/1

- The package Tapes provides the necessary declarations of the type and its primitive operations. 9
- Name_Server is a remote call interface package and is elaborated in a separate active partition to provide the necessary naming services (such as Register and Find) to the entire distributed program through remote subprogram calls. 10
- Tape_Driver is a normal package that is elaborated in a partition configured on the processing node that is connected to the tape device(s). The abstract operations are overridden to support the locally declared tape devices (Tape1, Tape2). The package is not visible to its clients, but it exports the tape devices (as remote objects) through the services of the Name_Server. This allows for tape devices to be dynamically added, removed or replaced without requiring the modification of the clients' code. 11
- {*AI12-0442-1*} The Tape_Client procedure references only declarations in the Tapes and Name_Server packages. Before using a tape for the first time, it will query the Name_Server for a system-wide identity for that tape. From then on, it can use that identity to access the tape device. 12/5
- Values of remote access type Tape_Ptr include the necessary information to complete the remote dispatching operations that result from dereferencing the controlling operands T1 and T2. 13

E.5 Partition Communication Subsystem

1/2 {*AI95-00273-01*} [The *Partition Communication Subsystem* (PCS) provides facilities for supporting communication between the active partitions of a distributed program. The package System.RPC is a language-defined interface to the PCS.]

1.a **Reason:** The prefix RPC is used rather than RSC because the term remote procedure call and its acronym are more familiar.

Static Semantics

2 The following language-defined library package exists:

3/5
```
{AI12-0241-1} {AI12-0302-1} with Ada.Streams; -- see 13.13.1
package System.RPC
   with Nonblocking => False, Global => in out synchronized is
```

4
```
   type Partition_Id is range 0 .. implementation-defined;
```

5
```
   Communication_Error : exception;
```

6
```
   type Params_Stream_Type (
      Initial_Size : Ada.Streams.Stream_Element_Count) is new
      Ada.Streams.Root_Stream_Type with private;
```

7
```
   procedure Read(
      Stream : in out Params_Stream_Type;
      Item : out Ada.Streams.Stream_Element_Array;
      Last : out Ada.Streams.Stream_Element_Offset);
```

8
```
   procedure Write(
      Stream : in out Params_Stream_Type;
      Item : in Ada.Streams.Stream_Element_Array);
```

9
```
   -- Synchronous call
   procedure Do_RPC(
      Partition  : in Partition_Id;
      Params     : access Params_Stream_Type;
      Result     : access Params_Stream_Type);
```

10
```
   -- Asynchronous call
   procedure Do_APC(
      Partition  : in Partition_Id;
      Params     : access Params_Stream_Type);
```

11
```
   -- The handler for incoming RPCs
   type RPC_Receiver is access procedure(
      Params     : access Params_Stream_Type;
      Result     : access Params_Stream_Type);
```

12
```
   procedure Establish_RPC_Receiver(
      Partition : in Partition_Id;
      Receiver  : in RPC_Receiver);
```

13
```
private
   ... -- not specified by the language
end System.RPC;
```

14 A value of the type Partition_Id is used to identify a partition.

14.a/2 **Implementation defined:** The range of type System.RPC.Partition_Id.

15 An object of the type Params_Stream_Type is used for identifying the particular remote subprogram that is being called, as well as marshalling and unmarshalling the parameters or result of a remote subprogram call, as part of sending them between partitions.

16 [The Read and Write procedures override the corresponding abstract operations for the type Params_Stream_Type.]

Dynamic Semantics

The Do_RPC and Do_APC procedures send a message to the active partition identified by the Partition parameter. 17

Implementation Note: It is assumed that the RPC interface is above the message-passing layer of the network protocol stack and is implemented in terms of it. 17.a

After sending the message, Do_RPC blocks the calling task until a reply message comes back from the called partition or some error is detected by the underlying communication system in which case Communication_Error is raised at the point of the call to Do_RPC. 18

Reason: Only one exception is defined in System.RPC, although many sources of errors might exist. This is so because it is not always possible to distinguish among these errors. In particular, it is often impossible to tell the difference between a failing communication link and a failing processing node. Additional information might be associated with a particular Exception_Occurrence for a Communication_Error. 18.a

Do_APC operates in the same way as Do_RPC except that it is allowed to return immediately after sending the message. 19

Upon normal return, the stream designated by the Result parameter of Do_RPC contains the reply message. 20

The procedure System.RPC.Establish_RPC_Receiver is called once, immediately after elaborating the library units of an active partition (that is, right after the *elaboration of the partition*) if the partition includes an RCI library unit, but prior to invoking the main subprogram, if any. The Partition parameter is the Partition_Id of the active partition being elaborated. The Receiver parameter designates an implementation-provided procedure called the *RPC-receiver* which will handle all RPCs received by the partition from the PCS. Establish_RPC_Receiver saves a reference to the RPC-receiver; when a message is received at the called partition, the RPC-receiver is called with the Params stream containing the message. When the RPC-receiver returns, the contents of the stream designated by Result is placed in a message and sent back to the calling partition. 21

Implementation Note: It is defined by the PCS implementation whether one or more threads of control should be available to process incoming messages and to wait for their completion. 21.a

Implementation Note: At link-time, the linker provides the RPC-receiver and the necessary tables to support it. A call on Establish_RPC_Receiver is inserted just before the call on the main subprogram. 21.b

Reason: The interface between the PCS (the System.RPC package) and the RPC-receiver is defined to be dynamic in order to allow the elaboration sequence to notify the PCS that all packages have been elaborated and that it is safe to call the receiving stubs. It is not guaranteed that the PCS units will be the last to be elaborated, so some other indication that elaboration is complete is needed. 21.c

If a call on Do_RPC is aborted, a cancellation message is sent to the called partition, to request that the execution of the remotely called subprogram be aborted. 22

To be honest: The full effects of this message are dependent on the implementation of the PCS. 22.a

This paragraph was deleted.{*AI12-0241-1*} 23/5

Implementation Requirements

The implementation of the RPC-receiver shall be reentrant[, thereby allowing concurrent calls on it from the PCS to service concurrent remote subprogram calls into the partition]. 24

Reason: There seems no reason to allow the implementation of RPC-receiver to be nonreentrant, even though we don't require that every implementation of the PCS actually perform concurrent calls on the RPC-receiver. 24.a

{*8652/0087*} {*AI95-00082-01*} An implementation shall not restrict the replacement of the body of System.RPC. An implementation shall not restrict children of System.RPC. [The related implementation permissions in the introduction to Annex A do not apply.] 24.1/1

24.a.1/1 **Reason:** The point of System.RPC is to let the user tailor the communications mechanism without requiring changes to or other cooperation from the compiler. However, implementations can restrict the replacement of language-defined units. This requirement overrides that permission for System.RPC.

24.2/1 {*8652/0087*} {*AI95-00082-01*} If the implementation of System.RPC is provided by the user, an implementation shall support remote subprogram calls as specified.

24.b/2 **Discussion:** {*AI95-00273-01*} If the implementation takes advantage of the implementation permission to use a different specification for System.RPC, it still needs to use it for remote subprogram calls, and allow the user to replace the body of System.RPC. It just isn't guaranteed to be portable to do so in Ada 2005 - an advantage which was more theoretical than real anyway.

Documentation Requirements

25 The implementation of the PCS shall document whether the RPC-receiver is invoked from concurrent tasks. If there is an upper limit on the number of such tasks, this limit shall be documented as well, together with the mechanisms to configure it (if this is supported).

25.a/2 *This paragraph was deleted.*

25.a.1/2 **Documentation Requirement:** Whether the RPC-receiver is invoked from concurrent tasks, and if so, the number of such tasks.

Implementation Permissions

26 The PCS is allowed to contain implementation-defined interfaces for explicit message passing, broadcasting, etc. Similarly, it is allowed to provide additional interfaces to query the state of some remote partition (given its partition ID) or of the PCS itself, to set timeouts and retry parameters, to get more detailed error status, etc. These additional interfaces should be provided in child packages of System.RPC.

26.a **Implementation defined:** Implementation-defined interfaces in the PCS.

27/5 {*AI12-0444-1*} A body for the package System.RPC is not required to be supplied by the implementation.

27.a **Reason:** It is presumed that a body for the package System.RPC might be extremely environment specific. Therefore, we do not require that a body be provided by the (compiler) implementation. The user will have to write a body, or acquire one, appropriate for the target environment.

27.1/3 {*AI95-00273-01*} {*AI05-0299-1*} An alternative declaration is allowed for package System.RPC as long as it provides a set of operations that is substantially equivalent to the specification defined in this subclause.

27.b/2 **Reason:** Experience has proved that the definition of System.RPC given here is inadequate for interfacing to existing distribution mechanisms (such as CORBA), especially on heterogeneous systems. Rather than mandate a change in the mechanism (which would break existing systems), require implementations to support multiple mechanisms (which is impractical), or prevent the use of Annex E facilities with existing systems (which would be silly), we simply make this facility optional.

27.c/2 One of the purposes behind System.RPC was that knowledgeable users, rather than compiler vendors, could create this package tailored to their networks. Experience has shown that users get their RPC from vendors anyway; users have not taken advantage of the flexibility provided by this defined interface. Moreover, one could compare this defined interface to requiring Ada compilers to use a defined interface to implement tasking. No one thinks that the latter is a good idea, why should anyone believe that the former is?

27.d/3 {*AI05-0299-1*} Therefore, this subclause is made optional. We considered deleting the subclause outright, but we still require that users may replace the package (whatever its interface). Also, it still provides a useful guide to the implementation of this feature.

Implementation Advice

28 Whenever possible, the PCS on the called partition should allow for multiple tasks to call the RPC-receiver with different messages and should allow them to block until the corresponding subprogram body returns.

28.a/2 **Implementation Advice:** The PCS should allow for multiple tasks to call the RPC-receiver.

The Write operation on a stream of type Params_Stream_Type should raise Storage_Error if it runs out of space trying to write the Item into the stream. 29

Implementation Advice: The System.RPC.Write operation should raise Storage_Error if it runs out of space when writing an item. 29.a.1/2

Implementation Note: An implementation could also dynamically allocate more space as needed, only propagating Storage_Error if the allocator it calls raises Storage_Error. This storage could be managed through a controlled component of the stream object, to ensure that it is reclaimed when the stream object is finalized. 29.a

NOTE The package System.RPC is not designed for direct calls by user programs. It is instead designed for use in the implementation of remote subprograms calls, being called by the calling stubs generated for a remote call interface library unit to initiate a remote call, and in turn calling back to an RPC-receiver that dispatches to the receiving stubs generated for the body of a remote call interface, to handle a remote call received from elsewhere. 30

Incompatibilities With Ada 95

{*AI95-00273-01*} The specification of System.RPC can now be tailored for an implementation. If a program replaces the body of System.RPC with a user-defined body, it might not compile in a given implementation of Ada 2005 (if the specification of System.RPC has been changed). 30.a/2

Wording Changes from Ada 95

{*8652/0087*} {*AI95-00082-01*} **Corrigendum:** Clarified that the user can replace System.RPC. 30.b/2

Annex F
(normative)
Information Systems

This Annex provides a set of facilities relevant to Information Systems programming. These fall into several categories: 1

- an attribute definition clause specifying Machine_Radix for a decimal subtype; 2
- the package Decimal, which declares a set of constants defining the implementation's capacity for decimal types, and a generic procedure for decimal division; and 3
- {*AI95-00285-01*} the child packages Text_IO.Editing, Wide_Text_IO.Editing, and Wide_Wide_Text_IO.Editing, which support formatted and localized output of decimal data, based on "picture String" values. 4/2

{*AI95-00434-01*} {*AI12-0449-1*} Other relevant facilities can be found in: 3.5.9, "Fixed Point Types"; 3.5.10, "Operations of Fixed Point Types"; 4.6, "Type Conversions"; 13.3, "Operational and Representation Attributes"; A.10.9, "Input-Output for Real Types"; B.3, "Interfacing with C and C++"; B.4, "Interfacing with COBOL"; Annex G, "Numerics". 5/5

The character and string handling packages in Annex A, "Predefined Language Environment" are also relevant for Information Systems. 6

Implementation Advice

{*AI05-0229-1*} If COBOL (respectively, C) is widely supported in the target environment, implementations supporting the Information Systems Annex should provide the child package Interfaces.COBOL (respectively, Interfaces.C) specified in Annex B and should support a *convention*_identifier of COBOL (respectively, C) for the Convention aspect (see Annex B), thus allowing Ada programs to interface with programs written in that language. 7/3

Implementation Advice: If COBOL (respectively, C) is supported in the target environment, then interfacing to COBOL (respectively, C) should be supported as specified in Annex B. 7.a/2

Extensions to Ada 83

This Annex is new to Ada 95. 7.b

Wording Changes from Ada 95

{*AI95-00285-01*} Added a mention of Wide_Wide_Text_IO.Editing, part of the support for 32-bit characters. 7.c/2

F.1 Machine_Radix Attribute Definition Clause

Static Semantics

{*AI12-0272-1*} The representation attribute Machine_Radix may be specified for a decimal first subtype (see 3.5.9) via an attribute_definition_clause; the expression of such a clause shall be static, and its value shall be 2 or 10. A value of 2 implies a binary base range; a value of 10 implies a decimal base range. 1/5

Ramification: In the absence of a Machine_Radix clause, the choice of 2 versus 10 for S'Machine_Radix is not specified. 1.a

Aspect Description for Machine_Radix: Radix (2 or 10) that is used to represent a decimal fixed point type. 1.b/3

Implementation Advice

2 Packed decimal should be used as the internal representation for objects of subtype S when S'Machine_Radix = 10.

2.a/2 **Implementation Advice:** Packed decimal should be used as the internal representation for objects of subtype *S* when *S*'Machine_Radix = 10.

2.b/3 **Discussion:** {*AI05-0229-1*} The intent of a decimal Machine_Radix attribute definition clause is to allow the programmer to declare an Ada decimal data object whose representation matches a particular COBOL implementation's representation of packed decimal items. The Ada object may then be passed to an interfaced COBOL program that takes a packed decimal data item as a parameter, assuming that convention COBOL has been specified for the Ada object's type with an aspect Convention.

2.c Additionally, the Ada compiler may choose to generate arithmetic instructions that exploit the packed decimal representation.

Examples

3 *Example of Machine_Radix attribute definition clause:*

4
```
type Money is delta 0.01 digits 15;
for Money'Machine_Radix use 10;
```

Wording Changes from Ada 2012

4.a/5 {*AI12-0272-1*} Clarified that Machine_Radix is a representation aspect.

F.2 The Package Decimal

Static Semantics

1 The library package Decimal has the following declaration:

2/5
```
{AI12-0414-1} package Ada.Decimal
   with Pure is

3     Max_Scale : constant := implementation-defined;
      Min_Scale : constant := implementation-defined;

4     Min_Delta : constant := 10.0**(-Max_Scale);
      Max_Delta : constant := 10.0**(-Min_Scale);

5     Max_Decimal_Digits : constant := implementation-defined;

6/3 {AI05-0229-1}    generic
         type Dividend_Type  is delta <> digits <>;
         type Divisor_Type   is delta <> digits <>;
         type Quotient_Type  is delta <> digits <>;
         type Remainder_Type is delta <> digits <>;
      procedure Divide (Dividend  : in Dividend_Type;
                        Divisor   : in Divisor_Type;
                        Quotient  : out Quotient_Type;
                        Remainder : out Remainder_Type)
         with Convention => Intrinsic;

7  end Ada.Decimal;
```

7.a **Implementation defined:** The values of named numbers in the package Decimal.

8 Max_Scale is the largest N such that 10.0**(–N) is allowed as a decimal type's delta. Its type is *universal_integer*.

9 Min_Scale is the smallest N such that 10.0**(–N) is allowed as a decimal type's delta. Its type is *universal_integer*.

10 Min_Delta is the smallest value allowed for *delta* in a decimal_fixed_point_definition. Its type is *universal_real*.

Max_Delta is the largest value allowed for *delta* in a decimal_fixed_point_definition. Its type is *universal_real*. 11

Max_Decimal_Digits is the largest value allowed for *digits* in a decimal_fixed_point_definition. Its type is *universal_integer*. 12

Reason: The name is Max_Decimal_Digits versus Max_Digits, in order to avoid confusion with the named number System.Max_Digits relevant to floating point. 12.a

Static Semantics

The effect of Divide is as follows. The value of Quotient is Quotient_Type(Dividend/Divisor). The value of Remainder is Remainder_Type(Intermediate), where Intermediate is the difference between Dividend and the product of Divisor and Quotient; this result is computed exactly. 13

Implementation Requirements

Decimal.Max_Decimal_Digits shall be at least 18. 14

Decimal.Max_Scale shall be at least 18. 15

Decimal.Min_Scale shall be at most 0. 16

NOTE The effect of division yielding a quotient with control over rounding versus truncation is obtained by applying either the function attribute Quotient_Type'Round or the conversion Quotient_Type to the expression Dividend/Divisor. 17

F.3 Edited Output for Decimal Types

{*AI95-00285-01*} The child packages Text_IO.Editing, Wide_Text_IO.Editing, and Wide_Wide_Text_IO.Editing provide localizable formatted text output, known as *edited output*, for decimal types. An edited output string is a function of a numeric value, program-specifiable locale elements, and a format control value. The numeric value is of some decimal type. The locale elements are: 1/2

- the currency string; 2
- the digits group separator character; 3
- the radix mark character; and 4
- the fill character that replaces leading zeros of the numeric value. 5

{*AI95-00285-01*} For Text_IO.Editing the edited output and currency strings are of type String, and the locale characters are of type Character. For Wide_Text_IO.Editing their types are Wide_String and Wide_Character, respectively. For Wide_Wide_Text_IO.Editing their types are Wide_Wide_String and Wide_Wide_Character, respectively. 6/2

Each of the locale elements has a default value that can be replaced or explicitly overridden. 7

A format-control value is of the private type Picture; it determines the composition of the edited output string and controls the form and placement of the sign, the position of the locale elements and the decimal digits, the presence or absence of a radix mark, suppression of leading zeros, and insertion of particular character values. 8

A Picture object is composed from a String value, known as a *picture String*, that serves as a template for the edited output string, and a Boolean value that controls whether a string of all space characters is produced when the number's value is zero. A picture String comprises a sequence of one- or two-Character symbols, each serving as a placeholder for a character or string at a corresponding position in the edited output string. The picture String symbols fall into several categories based on their effect on the edited output string: 9

10

Decimal Digit:	'9'					
Radix Control:	'.'	'V'				
Sign Control:	'+'	'–'	'<'	'>'	"CR"	"DB"
Currency Control:	'$'	'#'				
Zero Suppression:	'Z'	'*'				
Simple Insertion:	'_'	'B'	'0'	'/'		

11 The entries are not case-sensitive. Mixed- or lower-case forms for "CR" and "DB", and lower-case forms for 'V', 'Z', and 'B', have the same effect as the upper-case symbols shown.

12 An occurrence of a '9' Character in the picture String represents a decimal digit position in the edited output string.

13 A radix control Character in the picture String indicates the position of the radix mark in the edited output string: an actual character position for '.', or an assumed position for 'V'.

14 A sign control Character in the picture String affects the form of the sign in the edited output string. The '<' and '>' Character values indicate parentheses for negative values. A Character '+', '–', or '<' appears either singly, signifying a fixed-position sign in the edited output, or repeated, signifying a floating-position sign that is preceded by zero or more space characters and that replaces a leading 0.

15 A currency control Character in the picture String indicates an occurrence of the currency string in the edited output string. The '$' Character represents the complete currency string; the '#' Character represents one character of the currency string. A '$' Character appears either singly, indicating a fixed-position currency string in the edited output, or repeated, indicating a floating-position currency string that occurs in place of a leading 0. A sequence of '#' Character values indicates either a fixed- or floating-position currency string, depending on context.

16 A zero suppression Character in the picture String allows a leading zero to be replaced by either the space character (for 'Z') or the fill character (for '*').

17 A simple insertion Character in the picture String represents, in general, either itself (if '/' or '0'), the space character (if 'B'), or the digits group separator character (if '_'). In some contexts it is treated as part of a floating sign, floating currency, or zero suppression string.

18/2 {*AI95-00434-01*} An example of a picture String is "<###Z_ZZ9.99>". If the currency string is "kr", the separator character is ',', and the radix mark is '.' then the edited output string values for the decimal values 32.10 and –5432.10 are "bbkrbbb32.10b" and "(bkr5,432.10)", respectively, where 'b' indicates the space character.

19/2 {*AI95-00285-01*} The generic packages Text_IO.Decimal_IO, Wide_Text_IO.Decimal_IO, and Wide_-Wide_Text_IO.Decimal_IO (see A.10.9, “Input-Output for Real Types”) provide text input and nonedited text output for decimal types.

20/2 NOTE {*AI95-00285-01*} A picture String is of type Standard.String, for all of Text_IO.Editing, Wide_Text_IO.Editing, and Wide_Wide_Text_IO.Editing.

Wording Changes from Ada 95

20.a/2 {*AI95-00285-01*} Added descriptions of Wide_Wide_Text_IO.Editing; see F.3.5.

F.3.1 Picture String Formation

1/3 {*AI05-0299-1*} A *well-formed picture String*, or simply *picture String*, is a String value that conforms to the syntactic rules, composition constraints, and character replication conventions specified in this subclause.

Dynamic Semantics

This paragraph was deleted. 2/1

picture_string ::= 3
  fixed_$_picture_string
| fixed_#_picture_string
| floating_currency_picture_string
| non_currency_picture_string

fixed_$_picture_string ::= 4
  [fixed_LHS_sign] fixed_$_char {direct_insertion} [zero_suppression]
    number [RHS_sign]

| [fixed_LHS_sign {direct_insertion}] [zero_suppression]
    number fixed_$_char {direct_insertion} [RHS_sign]

| floating_LHS_sign number fixed_$_char {direct_insertion} [RHS_sign]

| [fixed_LHS_sign] fixed_$_char {direct_insertion}
    all_zero_suppression_number {direct_insertion} [RHS_sign]

| [fixed_LHS_sign {direct_insertion}] all_zero_suppression_number {direct_insertion}
    fixed_$_char {direct_insertion} [RHS_sign]

| all_sign_number {direct_insertion} fixed_$_char {direct_insertion} [RHS_sign]

fixed_#_picture_string ::= 5
  [fixed_LHS_sign] single_#_currency {direct_insertion}
    [zero_suppression] number [RHS_sign]

| [fixed_LHS_sign] multiple_#_currency {direct_insertion}
    zero_suppression number [RHS_sign]

| [fixed_LHS_sign {direct_insertion}] [zero_suppression]
    number fixed_#_currency {direct_insertion} [RHS_sign]

| floating_LHS_sign number fixed_#_currency {direct_insertion} [RHS_sign]

| [fixed_LHS_sign] single_#_currency {direct_insertion}
    all_zero_suppression_number {direct_insertion} [RHS_sign]

| [fixed_LHS_sign] multiple_#_currency {direct_insertion}
    all_zero_suppression_number {direct_insertion} [RHS_sign]

| [fixed_LHS_sign {direct_insertion}] all_zero_suppression_number {direct_insertion}
    fixed_#_currency {direct_insertion} [RHS_sign]

| all_sign_number {direct_insertion} fixed_#_currency {direct_insertion} [RHS_sign]

6 floating_currency_picture_string ::=
  [fixed_LHS_sign] {direct_insertion} floating_$_currency number [RHS_sign]
 | [fixed_LHS_sign] {direct_insertion} floating_#_currency number [RHS_sign]
 | [fixed_LHS_sign] {direct_insertion} all_currency_number {direct_insertion} [RHS_sign]

7 non_currency_picture_string ::=
  [fixed_LHS_sign {direct_insertion}] zero_suppression number [RHS_sign]
 | [floating_LHS_sign] number [RHS_sign]
 | [fixed_LHS_sign {direct_insertion}] all_zero_suppression_number {direct_insertion}
    [RHS_sign]
 | all_sign_number {direct_insertion}
 | fixed_LHS_sign direct_insertion {direct_insertion} number [RHS_sign]

8 fixed_LHS_sign ::= LHS_Sign

9 LHS_Sign ::= + | – | <

10 fixed_$_char ::= $

11 direct_insertion ::= simple_insertion

12 simple_insertion ::= _ | B | 0 | /

13 zero_suppression ::= Z {Z | context_sensitive_insertion} | fill_string

14 context_sensitive_insertion ::= simple_insertion

15 fill_string ::= * {* | context_sensitive_insertion}

16 number ::=
  fore_digits [radix [aft_digits] {direct_insertion}]
 | radix aft_digits {direct_insertion}

17 fore_digits ::= 9 {9 | direct_insertion}

18 aft_digits ::= {9 | direct_insertion} 9

19 radix ::= . | V

20 RHS_sign ::= + | – | > | CR | DB

21 floating_LHS_sign ::=
  LHS_Sign {context_sensitive_insertion} LHS_Sign {LHS_Sign | context_sensitive_insertion}

22 single_#_currency ::= #

23 multiple_#_currency ::= ## {#}

fixed_#_currency ::= single_#_currency | multiple_#_currency 24

floating_$_currency ::= 25
  $ {context_sensitive_insertion} $ {$ | context_sensitive_insertion}

floating_#_currency ::= 26
  # {context_sensitive_insertion} # {# | context_sensitive_insertion}

all_sign_number ::= all_sign_fore [radix [all_sign_aft]] [>] 27

all_sign_fore ::= 28
  sign_char {context_sensitive_insertion} sign_char {sign_char | context_sensitive_insertion}

all_sign_aft ::= {all_sign_aft_char} sign_char 29

all_sign_aft_char ::= sign_char | context_sensitive_insertion

sign_char ::= + | – | < 30

all_currency_number ::= all_currency_fore [radix [all_currency_aft]] 31

all_currency_fore ::= 32
  currency_char {context_sensitive_insertion}
    currency_char {currency_char | context_sensitive_insertion}

all_currency_aft ::= {all_currency_aft_char} currency_char 33

all_currency_aft_char ::= currency_char | context_sensitive_insertion

currency_char ::= $ | # 34

all_zero_suppression_number ::= all_zero_suppression_fore [radix [all_zero_suppression_aft]] 35

all_zero_suppression_fore ::= 36
  zero_suppression_char {zero_suppression_char | context_sensitive_insertion}

all_zero_suppression_aft ::= {all_zero_suppression_aft_char} zero_suppression_char 37

all_zero_suppression_aft_char ::= zero_suppression_char | context_sensitive_insertion

zero_suppression_char ::= Z | * 38

The following composition constraints apply to a picture String: 39

- A floating_LHS_sign does not have occurrences of different LHS_Sign Character values. 40
- If a picture String has '<' as fixed_LHS_sign, then it has '>' as RHS_sign. 41
- If a picture String has '<' in a floating_LHS_sign or in an all_sign_number, then it has an occurrence of '>'. 42
- {*8652/0088*} {*AI95-00153*} If a picture String has '+' or '–' as fixed_LHS_sign, in a floating_LHS_sign, or in an all_sign_number, then it has no RHS_sign or '>' character. 43/1
- An instance of all_sign_number does not have occurrences of different sign_char Character values. 44

45 • An instance of all_currency_number does not have occurrences of different currency_char Character values.

46 • An instance of all_zero_suppression_number does not have occurrences of different zero_-suppression_char Character values, except for possible case differences between 'Z' and 'z'.

47 A *replicable Character* is a Character that, by the above rules, can occur in two consecutive positions in a picture String.

48 A *Character replication* is a String

49 *char* & '(' & *spaces* & *count_string* & ')'

50 where *char* is a replicable Character, *spaces* is a String (possibly empty) comprising only space Character values, and *count_string* is a String of one or more decimal digit Character values. A Character replication in a picture String has the same effect as (and is said to be *equivalent to*) a String comprising *n* consecutive occurrences of *char*, where *n*=Integer'Value(*count_string*).

51 An *expanded picture String* is a picture String containing no Character replications.

51.a **Discussion:** Since 'B' is not allowed after a RHS sign, there is no need for a special rule to disallow "9.99DB(2)" as an abbreviation for "9.99DBB"

52 NOTE Although a sign to the left of the number can float, a sign to the right of the number is in a fixed position.

Wording Changes from Ada 95

52.a/2 {*8652/0088*} {*AI95-00153-01*} **Corrigendum:** The picture string rules for numbers were tightened.

F.3.2 Edited Output Generation

Dynamic Semantics

1 The contents of an edited output string are based on:

2 • A value, Item, of some decimal type Num,

3 • An expanded picture String Pic_String,

4 • A Boolean value, Blank_When_Zero,

5 • A Currency string,

6 • A Fill character,

7 • A Separator character, and

8 • A Radix_Mark character.

9 The combination of a True value for Blank_When_Zero and a '*' character in Pic_String is inconsistent; no edited output string is defined.

9.a/2 **Reason:** {*AI95-00114-01*} Such a Pic_String is invalid, and any attempt to use such a string will raise Picture_Error.

10 A layout error is identified in the rules below if leading nonzero digits of Item, character values of the Currency string, or a negative sign would be truncated; in such cases no edited output string is defined.

11 The edited output string has lower bound 1 and upper bound N where N = Pic_String'Length + Currency_Length_Adjustment – Radix_Adjustment, and

12 • Currency_Length_Adjustment = Currency'Length – 1 if there is some occurrence of '$' in Pic_String, and 0 otherwise.

13 • Radix_Adjustment = 1 if there is an occurrence of 'V' or 'v' in Pic_Str, and 0 otherwise.

Let the magnitude of Item be expressed as a base-10 number $I_p \cdots I_1.F_1 \cdots F_q$, called the *displayed magnitude* of Item, where: 14

- q = Min(Max(Num'Scale, 0), n) where n is 0 if Pic_String has no radix and is otherwise the number of digit positions following radix in Pic_String, where a digit position corresponds to an occurrence of '9', a zero_suppression_char (for an all_zero_suppression_number), a currency_char (for an all_currency_number), or a sign_char (for an all_sign_number). 15
- I_p /= 0 if p>0. 16

If n < Num'Scale, then the above number is the result of rounding (away from 0 if exactly midway between values). 17

If Blank_When_Zero = True and the displayed magnitude of Item is zero, then the edited output string comprises all space character values. Otherwise, the picture String is treated as a sequence of instances of syntactic categories based on the rules in F.3.1, and the edited output string is the concatenation of string values derived from these categories according to the following mapping rules. 18

{*AI12-0451-1*} Table F.1 shows the mapping from a sign control symbol to a corresponding character or string in the edited output. In the columns showing the edited output, a lower-case 'b' represents the space character. If there is no sign control symbol but the value of Item is negative, a layout error occurs and no edited output string is produced. 19/5

Table F.1: Edited Output for Sign Control Symbols

Sign Control Symbol	Edited Output for Nonnegative Number	Edited Output for Negative Number
'+'	'+'	'–'
'–'	'b'	'–'
'<'	'b'	'('
'>'	'b'	')'
"CR"	"bb"	"CR"
"DB"	"bb"	"DB"

{*AI12-0451-1*} An instance of fixed_LHS_sign maps to a character as shown in Table F.1. 20/5

An instance of fixed_$_char maps to Currency. 21

An instance of direct_insertion maps to Separator if direct_insertion = '_', and to the direct_insertion Character otherwise. 22

An instance of number maps to a string *integer_part* & *radix_part* & *fraction_part* where: 23

- The string for *integer_part* is obtained as follows: 24
  1. Occurrences of '9' in fore_digits of number are replaced from right to left with the decimal digit character values for I_1, ..., I_p, respectively. 25
  2. Each occurrence of '9' in fore_digits to the left of the leftmost '9' replaced according to rule 1 is replaced with '0'. 26
  3. If p exceeds the number of occurrences of '9' in fore_digits of number, then the excess leftmost digits are eligible for use in the mapping of an instance of zero_suppression, 27

floating_LHS_sign, floating_$_currency, or floating_#_currency to the left of number; if there is no such instance, then a layout error occurs and no edited output string is produced.

- The *radix_part* is:
  - "" if number does not include a radix, if radix = 'V', or if radix = 'v'
  - Radix_Mark if number includes '.' as radix
- The string for *fraction_part* is obtained as follows:
  1. Occurrences of '9' in aft_digits of number are replaced from left to right with the decimal digit character values for F_1, ... F_q.
  2. Each occurrence of '9' in aft_digits to the right of the rightmost '9' replaced according to rule 1 is replaced by '0'.

An instance of zero_suppression maps to the string obtained as follows:

1. The rightmost 'Z', 'z', or '*' Character values are replaced with the excess digits (if any) from the *integer_part* of the mapping of the number to the right of the zero_suppression instance,
2. A context_sensitive_insertion Character is replaced as though it were a direct_insertion Character, if it occurs to the right of some 'Z', 'z', or '*' in zero_suppression that has been mapped to an excess digit,
3. Each Character to the left of the leftmost Character replaced according to rule 1 above is replaced by:
   - the space character if the zero suppression Character is 'Z' or 'z', or
   - the Fill character if the zero suppression Character is '*'.
4. A layout error occurs if some excess digits remain after all 'Z', 'z', and '*' Character values in zero_suppression have been replaced via rule 1; no edited output string is produced.

{*AI12-0451-1*} An instance of RHS_sign maps to a character or string as shown in Table F.1.

An instance of floating_LHS_sign maps to the string obtained as follows.

1. Up to all but one of the rightmost LHS_Sign Character values are replaced by the excess digits (if any) from the *integer_part* of the mapping of the number to the right of the floating_LHS_sign instance.
2. {*AI12-0451-1*} The next Character to the left is replaced with the character given by the entry in Table F.1 corresponding to the LHS_Sign Character.
3. A context_sensitive_insertion Character is replaced as though it were a direct_insertion Character, if it occurs to the right of the leftmost LHS_Sign character replaced according to rule 1.
4. Any other Character is replaced by the space character.
5. A layout error occurs if some excess digits remain after replacement via rule 1; no edited output string is produced.

An instance of fixed_#_currency maps to the Currency string with n space character values concatenated on the left (if the instance does not follow a radix) or on the right (if the instance does follow a radix), where n is the difference between the length of the fixed_#_currency instance and Currency'Length. A layout error occurs if Currency'Length exceeds the length of the fixed_#_currency instance; no edited output string is produced.

An instance of floating_$_currency maps to the string obtained as follows:

1. Up to all but one of the rightmost '$' Character values are replaced with the excess digits (if any) from the *integer_part* of the mapping of the number to the right of the floating_$_currency instance. 50
2. The next Character to the left is replaced by the Currency string. 51
3. A context_sensitive_insertion Character is replaced as though it were a direct_insertion Character, if it occurs to the right of the leftmost '$' Character replaced via rule 1. 52
4. Each other Character is replaced by the space character. 53
5. A layout error occurs if some excess digits remain after replacement by rule 1; no edited output string is produced. 54

An instance of floating_#_currency maps to the string obtained as follows: 55

1. Up to all but one of the rightmost '#' Character values are replaced with the excess digits (if any) from the *integer_part* of the mapping of the number to the right of the floating_#_currency instance. 56
2. The substring whose last Character occurs at the position immediately preceding the leftmost Character replaced via rule 1, and whose length is Currency'Length, is replaced by the Currency string. 57
3. A context_sensitive_insertion Character is replaced as though it were a direct_insertion Character, if it occurs to the right of the leftmost '#' replaced via rule 1. 58
4. Any other Character is replaced by the space character. 59
5. A layout error occurs if some excess digits remain after replacement rule 1, or if there is no substring with the required length for replacement rule 2; no edited output string is produced. 60

An instance of all_zero_suppression_number maps to: 61

- a string of all spaces if the displayed magnitude of Item is zero, the zero_suppression_char is 'Z' or 'z', and the instance of all_zero_suppression_number does not have a radix at its last character position; 62
- a string containing the Fill character in each position except for the character (if any) corresponding to radix, if zero_suppression_char = '*' and the displayed magnitude of Item is zero; 63
- otherwise, the same result as if each zero_suppression_char in all_zero_suppression_aft were '9', interpreting the instance of all_zero_suppression_number as either zero_suppression number (if a radix and all_zero_suppression_aft are present), or as zero_suppression otherwise. 64

An instance of all_sign_number maps to: 65

- a string of all spaces if the displayed magnitude of Item is zero and the instance of all_sign_number does not have a radix at its last character position; 66
- otherwise, the same result as if each sign_char in all_sign_number_aft were '9', interpreting the instance of all_sign_number as either floating_LHS_sign number (if a radix and all_sign_number_aft are present), or as floating_LHS_sign otherwise. 67

An instance of all_currency_number maps to: 68

- a string of all spaces if the displayed magnitude of Item is zero and the instance of all_currency_number does not have a radix at its last character position; 69
- otherwise, the same result as if each currency_char in all_currency_number_aft were '9', interpreting the instance of all_currency_number as floating_$_currency number or 70

floating_#_currency number (if a radix and all_currency_number_aft are present), or as floating_$_currency or floating_#_currency otherwise.

Examples

71/5 {*AI12-0429-1*} *Examples of use of edited output; in the result string values shown below, 'b' represents the space character:*

```
72      Item:          Picture and Result Strings:
73/3    {AI05-0248-1} 123456.78     Picture:  "-###**_***_**9.99"
                      Result:   "bbb$***123,456.78"
                                "bbFF***123.456,78" (currency = "FF",
                                                     separator = '.',
                                                     radix mark = ',')
74/1    {8652/0089} {AI95-00070} 123456.78     Picture:  "-$**_***_**9.99"
                      Result:   "b$***123,456.78"
                               "bFF***123.456,78" (currency = "FF",
                                                   separator = '.',
                                                   radix mark = ',')
75      0.0           Picture: "-$$$$$$.$$"
                      Result:  "bbbbbbbbbb"
76      0.20          Picture: "-$$$$$$.$$"
                      Result:  "bbbbbb$.20"
77      -1234.565     Picture: "<<<<_<<<.<<###>"
                      Result:  "bb(1,234.57DMb)"  (currency = "DM")
78      12345.67      Picture: "###_###_##9.99"
                      Result:  "bbCHF12,345.67"   (currency = "CHF")
```

Wording Changes from Ada 95

78.a/2 {*8652/0089*} {*AI95-00070-01*} **Corrigendum:** Corrected the picture string example.

F.3.3 The Package Text_IO.Editing

1 The package Text_IO.Editing provides a private type Picture with associated operations, and a generic package Decimal_Output. An object of type Picture is composed from a well-formed picture String (see F.3.1) and a Boolean item indicating whether a zero numeric value will result in an edited output string of all space characters. The package Decimal_Output contains edited output subprograms implementing the effects defined in F.3.2.

Static Semantics

2 The library package Text_IO.Editing has the following declaration:

```
3/5     {AI12-0241-1} {AI12-0302-1} package Ada.Text_IO.Editing
           with Nonblocking, Global => in out synchronized is
4          type Picture is private;
5          function Valid (Pic_String      : in String;
                           Blank_When_Zero : in Boolean := False) return Boolean;
6          function To_Picture (Pic_String      : in String;
                                Blank_When_Zero : in Boolean := False)
              return Picture;
7          function Pic_String      (Pic : in Picture) return String;
           function Blank_When_Zero (Pic : in Picture) return Boolean;
8          Max_Picture_Length  : constant := implementation_defined;
9          Picture_Error       : exception;
```

```
   Default_Currency    : constant String    := "$";                                    10
   Default_Fill        : constant Character := '*';
   Default_Separator   : constant Character := ',';
   Default_Radix_Mark  : constant Character := '.';

   generic                                                                             11
      type Num is delta <> digits <>;
      Default_Currency   : in String    := Text_IO.Editing.Default_Currency;
      Default_Fill       : in Character := Text_IO.Editing.Default_Fill;
      Default_Separator  : in Character :=
                              Text_IO.Editing.Default_Separator;
      Default_Radix_Mark : in Character :=
                              Text_IO.Editing.Default_Radix_Mark;
   package Decimal_Output is
      function Length (Pic      : in Picture;
                       Currency : in String := Default_Currency)
         return Natural;

      function Valid (Item     : in Num;                                             12
                      Pic      : in Picture;
                      Currency : in String := Default_Currency)
         return Boolean;

      function Image (Item       : in Num;                                           13
                      Pic        : in Picture;
                      Currency   : in String    := Default_Currency;
                      Fill       : in Character := Default_Fill;
                      Separator  : in Character := Default_Separator;
                      Radix_Mark : in Character := Default_Radix_Mark)
         return String;

{AI12-0241-1}       procedure Put (File       : in File_Type;                       14/5
                       Item       : in Num;
                       Pic        : in Picture;
                       Currency   : in String    := Default_Currency;
                       Fill       : in Character := Default_Fill;
                       Separator  : in Character := Default_Separator;
                       Radix_Mark : in Character := Default_Radix_Mark)
         with Nonblocking => False;

{AI12-0241-1}       procedure Put (Item       : in Num;                             15/5
                       Pic        : in Picture;
                       Currency   : in String    := Default_Currency;
                       Fill       : in Character := Default_Fill;
                       Separator  : in Character := Default_Separator;
                       Radix_Mark : in Character := Default_Radix_Mark)
         with Nonblocking => False;

      procedure Put (To         : out String;                                        16
                     Item       : in Num;
                     Pic        : in Picture;
                     Currency   : in String    := Default_Currency;
                     Fill       : in Character := Default_Fill;
                     Separator  : in Character := Default_Separator;
                     Radix_Mark : in Character := Default_Radix_Mark);
   end Decimal_Output;
private
   ... -- not specified by the language
end Ada.Text_IO.Editing;
```

Implementation defined: The value of Max_Picture_Length in the package Text_IO.Editing 16.a

The exception Constraint_Error is raised if the Image function or any of the Put procedures is invoked with a null string for Currency. 17

```
function Valid (Pic_String      : in String;
                Blank_When_Zero : in Boolean := False) return Boolean;
```

Valid returns True if Pic_String is a well-formed picture String (see F.3.1) the length of whose expansion does not exceed Max_Picture_Length, and if either Blank_When_Zero is False or Pic_String contains no '*'.

```
function To_Picture (Pic_String      : in String;
                     Blank_When_Zero : in Boolean := False)
   return Picture;
```

To_Picture returns a result Picture such that the application of the function Pic_String to this result yields an expanded picture String equivalent to Pic_String, and such that Blank_When_Zero applied to the result Picture is the same value as the parameter Blank_When_Zero. Picture_Error is raised if not Valid(Pic_String, Blank_When_Zero).

```
function Pic_String      (Pic : in Picture) return String;

function Blank_When_Zero (Pic : in Picture) return Boolean;
```

If Pic is To_Picture(String_Item, Boolean_Item) for some String_Item and Boolean_Item, then:

- Pic_String(Pic) returns an expanded picture String equivalent to String_Item and with any lower-case letter replaced with its corresponding upper-case form, and
- Blank_When_Zero(Pic) returns Boolean_Item.

If Pic_1 and Pic_2 are objects of type Picture, then "="(Pic_1, Pic_2) is True when

- Pic_String(Pic_1) = Pic_String(Pic_2), and
- Blank_When_Zero(Pic_1) = Blank_When_Zero(Pic_2).

```
function Length (Pic      : in Picture;
                 Currency : in String := Default_Currency)
   return Natural;
```

Length returns Pic_String(Pic)'Length + Currency_Length_Adjustment – Radix_Adjustment where

- Currency_Length_Adjustment =
  - Currency'Length – 1 if there is some occurrence of '$' in Pic_String(Pic), and
  - 0 otherwise.
- Radix_Adjustment =
  - 1 if there is an occurrence of 'V' or 'v' in Pic_Str(Pic), and
  - 0 otherwise.

```
function Valid (Item     : in Num;
                Pic      : in Picture;
                Currency : in String := Default_Currency)
   return Boolean;
```

Valid returns True if Image(Item, Pic, Currency) does not raise Layout_Error, and returns False otherwise.

```
function Image (Item       : in Num;
                Pic        : in Picture;
                Currency   : in String    := Default_Currency;
                Fill       : in Character := Default_Fill;
                Separator  : in Character := Default_Separator;
                Radix_Mark : in Character := Default_Radix_Mark)
   return String;
```

Image returns the edited output String as defined in F.3.2 for Item, Pic_String(Pic), Blank_When_Zero(Pic), Currency, Fill, Separator, and Radix_Mark. If these rules identify a layout error, then Image raises the exception Layout_Error. 40

```
procedure Put (File       : in File_Type;                                    41
               Item       : in Num;
               Pic        : in Picture;
               Currency   : in String    := Default_Currency;
               Fill       : in Character := Default_Fill;
               Separator  : in Character := Default_Separator;
               Radix_Mark : in Character := Default_Radix_Mark);

procedure Put (Item       : in Num;
               Pic        : in Picture;
               Currency   : in String    := Default_Currency;
               Fill       : in Character := Default_Fill;
               Separator  : in Character := Default_Separator;
               Radix_Mark : in Character := Default_Radix_Mark);
```

Each of these Put procedures outputs Image(Item, Pic, Currency, Fill, Separator, Radix_Mark) consistent with the conventions for Put for other real types in case of bounded line length (see A.10.6, “Get and Put Procedures”). 42

```
procedure Put (To         : out String;                                      43
               Item       : in Num;
               Pic        : in Picture;
               Currency   : in String    := Default_Currency;
               Fill       : in Character := Default_Fill;
               Separator  : in Character := Default_Separator;
               Radix_Mark : in Character := Default_Radix_Mark);
```

{*AI05-0264-1*} Put copies Image(Item, Pic, Currency, Fill, Separator, Radix_Mark) to the given string, right justified. Otherwise, unassigned Character values in To are assigned the space character. If To'Length is less than the length of the string resulting from Image, then Layout_Error is raised. 44/3

Implementation Requirements

Max_Picture_Length shall be at least 30. The implementation shall support currency strings of length up to at least 10, both for Default_Currency in an instantiation of Decimal_Output, and for Currency in an invocation of Image or any of the Put procedures. 45

Discussion: This implies that a picture string with character replications need not be supported (i.e., To_Picture will raise Picture_Error) if its expanded form exceeds 30 characters. 45.a

Discussion: {*AI12-0450-1*} The rules for edited output were based on the 1985 version of COBOL (ANSI X3.23:1985, endorsed by ISO as ISO 1989-1985), with the following differences: 52.a/5

- The COBOL provisions for picture string localization and for 'P' format are absent from Ada. 52.b/5
- The following Ada facilities are not in the 1985 version of COBOL: 52.c/5
  - currency symbol placement after the number; 52.d/5
  - localization of edited output string for multi-character currency string values, including support for both length-preserving and length-expanding currency symbols in picture strings; 52.e/5
  - localization of the radix mark, digits separator, and fill character; and 52.f/5
  - parenthesization of negative values. 52.g/5

{*AI12-0450-1*} The value of 30 for Max_Picture_Length is the same limit as in COBOL. 52.h/5

Reason: There are several reasons we have not adopted the COBOL-style permission to provide a single-character replacement in the picture string for the `$' as currency symbol, or to interchange the roles of `.' and `,' in picture strings 52.i

- It would have introduced considerable complexity into Ada, as well as confusion between run-time and compile-time character interpretation, since picture Strings are dynamically computable in Ada, in contrast with COBOL 52.j

52.k
- Ada's rules for real literals provide a natural interpretation of `_' as digits separator and `.' for radix mark; it is not essential to allow these to be localized in picture strings, since Ada does not allow them to be localized in real literals.

52.l
- The COBOL restriction for the currency symbol in a picture string to be replaced by a single character currency symbol is a compromise solution. For general international usage a mechanism is needed to localize the edited output to be a multi-character currency string. Allowing a single-Character localization for the picture Character, and a multiple-character localization for the currency string, would be an unnecessary complication.

F.3.4 The Package Wide_Text_IO.Editing

Static Semantics

1 The child package Wide_Text_IO.Editing has the same contents as Text_IO.Editing, except that:

2
- each occurrence of Character is replaced by Wide_Character,

3
- each occurrence of Text_IO is replaced by Wide_Text_IO,

4
- the subtype of Default_Currency is Wide_String rather than String, and

5
- each occurrence of String in the generic package Decimal_Output is replaced by Wide_String.

5.a **Implementation defined:** The value of Max_Picture_Length in the package Wide_Text_IO.Editing

6 NOTE Each of the functions Wide_Text_IO.Editing.Valid, To_Picture, and Pic_String has String (versus Wide_String) as its parameter or result subtype, since a picture String is not localizable.

F.3.5 The Package Wide_Wide_Text_IO.Editing

Static Semantics

1/2 {*AI95-00285-01*} The child package Wide_Wide_Text_IO.Editing has the same contents as Text_IO.Editing, except that:

2/2
- each occurrence of Character is replaced by Wide_Wide_Character,

3/2
- each occurrence of Text_IO is replaced by Wide_Wide_Text_IO,

4/2
- the subtype of Default_Currency is Wide_Wide_String rather than String, and

5/2
- each occurrence of String in the generic package Decimal_Output is replaced by Wide_Wide_String.

5.a/2 **Implementation defined:** The value of Max_Picture_Length in the package Wide_Wide_Text_IO.Editing

6/2 NOTE {*AI95-00285-01*} Each of the functions Wide_Wide_Text_IO.Editing.Valid, To_Picture, and Pic_String has String (versus Wide_Wide_String) as its parameter or result subtype, since a picture String is not localizable.

Extensions to Ada 95

6.a/2 {*AI95-00285-01*} Package Wide_Wide_Text_IO.Editing is new; it supports 32-bit character strings. (Shouldn't it have been “Widest_Text_IO.Editing”? :-)

Annex G
(normative)
Numerics

The Numerics Annex specifies 1

- features for complex arithmetic, including complex I/O; 2
- a mode ("strict mode"), in which the predefined arithmetic operations of floating point and fixed point types and the functions and operations of various predefined packages have to provide guaranteed accuracy or conform to other numeric performance requirements, which the Numerics Annex also specifies; 3
- {*AI12-0445-1*} a mode ("relaxed mode"), in which there are no accuracy or other numeric performance requirements to be satisfied, as for implementations not conforming to the Numerics Annex; 4/5
- {*AI95-00296-01*} models of floating point and fixed point arithmetic on which the accuracy requirements of strict mode are based; 5/2
- {*AI95-00296-01*} the definitions of the model-oriented attributes of floating point types that apply in the strict mode; and 6/2
- {*AI95-00296-01*} features for the manipulation of real and complex vectors and matrices. 6.1/2

Implementation Advice

{*AI05-0229-1*} If Fortran (respectively, C) is widely supported in the target environment, implementations supporting the Numerics Annex should provide the child package Interfaces.Fortran (respectively, Interfaces.C) specified in Annex B and should support a *convention*_identifier of Fortran (respectively, C) for the Convention aspect (see Annex B), thus allowing Ada programs to interface with programs written in that language. 7/3

Implementation Advice: If Fortran (respectively, C) is supported in the target environment, then interfacing to Fortran (respectively, C) should be supported as specified in Annex B. 7.a.1/2

Extensions to Ada 83

This Annex is new to Ada 95. 7.a

G.1 Complex Arithmetic

Types and arithmetic operations for complex arithmetic are provided in Generic_Complex_Types, which is defined in G.1.1. Implementation-defined approximations to the complex analogs of the mathematical functions known as the "elementary functions" are provided by the subprograms in Generic_Complex_-Elementary_Functions, which is defined in G.1.2. Both of these library units are generic children of the predefined package Numerics (see A.5). Nongeneric equivalents of these generic packages for each of the predefined floating point types are also provided as children of Numerics. 1

Implementation defined: The accuracy actually achieved by the complex elementary functions and by other complex arithmetic operations. 1.a

Discussion: Complex arithmetic is defined in the Numerics Annex, rather than in the core, because it is considered to be a specialized need of (some) numeric applications. 1.b

G.1.1 Complex Types

Static Semantics

1 The generic library package Numerics.Generic_Complex_Types has the following declaration:

2/5
```
{8652/0020} {AI95-00126-01} {AI12-0241-1} generic
   type Real is digits <>;
package Ada.Numerics.Generic_Complex_Types
   with Pure, Nonblocking is
```

3
```
   type Complex is
      record
         Re, Im : Real'Base;
      end record;
```

4/5
```
{AI95-00161-01} {AI95-0399-1}    type Imaginary is private
      with Preelaborable_Initialization;
```

5
```
   i : constant Imaginary;
   j : constant Imaginary;
```

6
```
   function Re (X : Complex)   return Real'Base;
   function Im (X : Complex)   return Real'Base;
   function Im (X : Imaginary) return Real'Base;
```

7
```
   procedure Set_Re (X  : in out Complex;
                     Re : in     Real'Base);
   procedure Set_Im (X  : in out Complex;
                     Im : in     Real'Base);
   procedure Set_Im (X  :    out Imaginary;
                     Im : in     Real'Base);
```

8
```
   function Compose_From_Cartesian (Re, Im : Real'Base) return Complex;
   function Compose_From_Cartesian (Re     : Real'Base) return Complex;
   function Compose_From_Cartesian (Im     : Imaginary) return Complex;
```

9
```
   function Modulus (X     : Complex) return Real'Base;
   function "abs"   (Right : Complex) return Real'Base renames Modulus;
```

10
```
   function Argument (X     : Complex)   return Real'Base;
   function Argument (X     : Complex;
                      Cycle : Real'Base) return Real'Base;
```

11
```
   function Compose_From_Polar (Modulus, Argument        : Real'Base)
      return Complex;
   function Compose_From_Polar (Modulus, Argument, Cycle : Real'Base)
      return Complex;
```

12
```
   function "+"       (Right : Complex) return Complex;
   function "-"       (Right : Complex) return Complex;
   function Conjugate (X     : Complex) return Complex;
```

13
```
   function "+" (Left, Right : Complex) return Complex;
   function "-" (Left, Right : Complex) return Complex;
   function "*" (Left, Right : Complex) return Complex;
   function "/" (Left, Right : Complex) return Complex;
```

14
```
   function "**" (Left : Complex; Right : Integer) return Complex;
```

15
```
   function "+"       (Right : Imaginary) return Imaginary;
   function "-"       (Right : Imaginary) return Imaginary;
   function Conjugate (X     : Imaginary) return Imaginary renames "-";
   function "abs"     (Right : Imaginary) return Real'Base;
```

16
```
   function "+" (Left, Right : Imaginary) return Imaginary;
   function "-" (Left, Right : Imaginary) return Imaginary;
   function "*" (Left, Right : Imaginary) return Real'Base;
   function "/" (Left, Right : Imaginary) return Real'Base;
```

17
```
   function "**" (Left : Imaginary; Right : Integer) return Complex;
```

```
   function "<"  (Left, Right : Imaginary) return Boolean;                                    18
   function "<=" (Left, Right : Imaginary) return Boolean;
   function ">"  (Left, Right : Imaginary) return Boolean;
   function ">=" (Left, Right : Imaginary) return Boolean;

   function "+" (Left : Complex;   Right : Real'Base) return Complex;                         19
   function "+" (Left : Real'Base; Right : Complex)   return Complex;
   function "-" (Left : Complex;   Right : Real'Base) return Complex;
   function "-" (Left : Real'Base; Right : Complex)   return Complex;
   function "*" (Left : Complex;   Right : Real'Base) return Complex;
   function "*" (Left : Real'Base; Right : Complex)   return Complex;
   function "/" (Left : Complex;   Right : Real'Base) return Complex;
   function "/" (Left : Real'Base; Right : Complex)   return Complex;

   function "+" (Left : Complex;   Right : Imaginary) return Complex;                         20
   function "+" (Left : Imaginary; Right : Complex)   return Complex;
   function "-" (Left : Complex;   Right : Imaginary) return Complex;
   function "-" (Left : Imaginary; Right : Complex)   return Complex;
   function "*" (Left : Complex;   Right : Imaginary) return Complex;
   function "*" (Left : Imaginary; Right : Complex)   return Complex;
   function "/" (Left : Complex;   Right : Imaginary) return Complex;
   function "/" (Left : Imaginary; Right : Complex)   return Complex;

   function "+" (Left : Imaginary; Right : Real'Base) return Complex;                         21
   function "+" (Left : Real'Base; Right : Imaginary) return Complex;
   function "-" (Left : Imaginary; Right : Real'Base) return Complex;
   function "-" (Left : Real'Base; Right : Imaginary) return Complex;
   function "*" (Left : Imaginary; Right : Real'Base) return Imaginary;
   function "*" (Left : Real'Base; Right : Imaginary) return Imaginary;
   function "/" (Left : Imaginary; Right : Real'Base) return Imaginary;
   function "/" (Left : Real'Base; Right : Imaginary) return Imaginary;

private                                                                                       22

   type Imaginary is new Real'Base;                                                           23
   i : constant Imaginary := 1.0;
   j : constant Imaginary := 1.0;

end Ada.Numerics.Generic_Complex_Types;                                                       24
```

{*8652/0020*} {*AI95-00126-01*} The library package Numerics.Complex_Types is declared pure and defines the same types, constants, and subprograms as Numerics.Generic_Complex_Types, except that the predefined type Float is systematically substituted for Real'Base throughout. Nongeneric equivalents of Numerics.Generic_Complex_Types for each of the other predefined floating point types are defined similarly, with the names Numerics.Short_Complex_Types, Numerics.Long_Complex_Types, etc. 25/1

Reason: The nongeneric equivalents are provided to allow the programmer to construct simple mathematical applications without being required to understand and use generics. 25.a

Reason: The nongeneric equivalents all export the types Complex and Imaginary and the constants i and j (rather than uniquely named types and constants, such as Short_Complex, Long_Complex, etc.) to preserve their equivalence to actual instantiations of the generic package and to allow the programmer to change the precision of an application globally by changing a single context clause. 25.b

Ramification: {*AI12-0241-1*} A pure unit is always nonblocking, so we don't need to talk about the Nonblocking aspect of these packages. Similarly, a pure unit always has the Global aspect set to **null**, so we don't need to talk about that, either. 25.c/5

{*AI95-00434-01*} [Complex is a visible type with Cartesian components.] 26/2

Reason: The Cartesian representation is far more common than the polar representation, in practice. The accuracy of the results of the complex arithmetic operations and of the complex elementary functions is dependent on the representation; thus, implementers need to know that representation. The type is visible so that complex “literals” can be written in aggregate notation, if desired. 26.a

[Imaginary is a private type; its full type is derived from Real'Base.] 27

Reason: The Imaginary type and the constants i and j are provided for two reasons: 27.a

- They allow complex “literals” to be written in the alternate form of *a* + *b**i (or *a* + *b**j), if desired. Of course, in some contexts the sum will need to be parenthesized. 27.b

27.c/5 • {*AI12-0437-1*} {*AI12-0453-1*} When an Ada binding to ISO/IEC 60559:2020 that provides (signed) infinities as the result of operations that overflow becomes available, it will be important to allow arithmetic between pure-imaginary and complex operands without requiring the former to be represented as (or promoted to) complex values with a real component of zero. For example, the multiplication of *a* + *b**i by *d**i should yield –*b*· *d* + *a*· *d**i, but if one cannot avoid representing the pure-imaginary value *d**i as the complex value 0.0 + *d**i, then a NaN ("Not-a-Number") could be produced as the result of multiplying *a* by 0.0 (for example, when *a* is infinite); the NaN could later trigger an exception. Providing the Imaginary type and overloadings of the arithmetic operators for mixtures of Imaginary and Complex operands gives the programmer the same control over avoiding premature coercion of pure-imaginary values to complex as is already provided for pure-real values.

27.d **Reason:** The Imaginary type is private, rather than being visibly derived from Real'Base, for two reasons:

27.e • to preclude implicit conversions of real literals to the Imaginary type (such implicit conversions would make many common arithmetic expressions ambiguous); and

27.f • to suppress the implicit derivation of the multiplication, division, and absolute value operators with Imaginary operands and an Imaginary result (the result type would be incorrect).

27.g/5 **Reason:** {*AI12-0453-1*} The base subtype Real'Base is used for the component type of Complex, the parent type of Imaginary, and the parameter and result types of some of the subprograms to maximize the chances of being able to pass meaningful values into the subprograms and receive meaningful results back. The generic formal parameter Real therefore plays only one role, that of providing the precision to be maintained in complex arithmetic calculations. Thus, the subprograms in Numerics.Generic_Complex_Types share with those in Numerics.Generic_Elementary_Functions, and indeed even with the predefined arithmetic operations (see 4.5), the property of being free of range checks on input and output, that is, of being able to exploit the base range of the relevant floating point type fully. As a result, the user loses the ability to impose application-oriented bounds on the range of values that the components of a complex variable can acquire; however, it can be argued that few, if any, applications have a naturally square domain (as opposed to a circular domain) anyway.

28 The arithmetic operations and the Re, Im, Modulus, Argument, and Conjugate functions have their usual mathematical meanings. When applied to a parameter of pure-imaginary type, the “imaginary-part” function Im yields the value of its parameter, as the corresponding real value. The remaining subprograms have the following meanings:

28.a **Reason:** The middle case can be understood by considering the parameter of pure-imaginary type to represent a complex value with a zero real part.

29 • The Set_Re and Set_Im procedures replace the designated component of a complex parameter with the given real value; applied to a parameter of pure-imaginary type, the Set_Im procedure replaces the value of that parameter with the imaginary value corresponding to the given real value.

30 • The Compose_From_Cartesian function constructs a complex value from the given real and imaginary components. If only one component is given, the other component is implicitly zero.

31 • The Compose_From_Polar function constructs a complex value from the given modulus (radius) and argument (angle). When the value of the parameter Modulus is positive (resp., negative), the result is the complex value represented by the point in the complex plane lying at a distance from the origin given by the absolute value of Modulus and forming an angle measured counterclockwise from the positive (resp., negative) real axis given by the value of the parameter Argument.

32 When the Cycle parameter is specified, the result of the Argument function and the parameter Argument of the Compose_From_Polar function are measured in units such that a full cycle of revolution has the given value; otherwise, they are measured in radians.

33 The computed results of the mathematically multivalued functions are rendered single-valued by the following conventions, which are meant to imply the principal branch:

34 • The result of the Modulus function is nonnegative.

35 • The result of the Argument function is in the quadrant containing the point in the complex plane represented by the parameter X. This may be any quadrant (I through IV); thus, the range of the Argument function is approximately –π to π (–Cycle/2.0 to Cycle/2.0, if the parameter Cycle is

specified). When the point represented by the parameter X lies on the negative real axis, the result approximates

- π (resp., –π) when the sign of the imaginary component of X is positive (resp., negative), if Real'Signed_Zeros is True; 36
- π, if Real'Signed_Zeros is False. 37

- Because a result lying on or near one of the axes may not be exactly representable, the approximation inherent in computing the result may place it in an adjacent quadrant, close to but on the wrong side of the axis. 38

Dynamic Semantics

The exception Numerics.Argument_Error is raised by the Argument and Compose_From_Polar functions with specified cycle, signaling a parameter value outside the domain of the corresponding mathematical function, when the value of the parameter Cycle is zero or negative. 39

The exception Constraint_Error is raised by the division operator when the value of the right operand is zero, and by the exponentiation operator when the value of the left operand is zero and the value of the exponent is negative, provided that Real'Machine_Overflows is True; when Real'Machine_Overflows is False, the result is unspecified. [Constraint_Error can also be raised when a finite result overflows (see G.2.6).] 40

Discussion: {*AI12-0437-1*} It is anticipated that an Ada binding to ISO/IEC 60559:2020 will be developed in the future. As part of such a binding, the Machine_Overflows attribute of a conformant floating point type will be specified to yield False, which will permit implementations of the complex arithmetic operations to deliver results with an infinite component (and set the overflow flag defined by the binding) instead of raising Constraint_Error in overflow situations, when traps are disabled. Similarly, it is appropriate for the complex arithmetic operations to deliver results with infinite components (and set the zero-divide flag defined by the binding) instead of raising Constraint_Error in the situations defined above, when traps are disabled. Finally, such a binding should also specify the behavior of the complex arithmetic operations, when sensible, given operands with infinite components. 40.a/5

Implementation Requirements

In the implementation of Numerics.Generic_Complex_Types, the range of intermediate values allowed during the calculation of a final result shall not be affected by any range constraint of the subtype Real. 41

Implementation Note: Implementations of Numerics.Generic_Complex_Types written in Ada should therefore avoid declaring local variables of subtype Real; the subtype Real'Base should be used instead. 41.a

In the following cases, evaluation of a complex arithmetic operation shall yield the *prescribed result*, provided that the preceding rules do not call for an exception to be raised: 42

- The results of the Re, Im, and Compose_From_Cartesian functions are exact. 43
- The real (resp., imaginary) component of the result of a binary addition operator that yields a result of complex type is exact when either of its operands is of pure-imaginary (resp., real) type. 44

Ramification: The result of the addition operator is exact when one of its operands is of real type and the other is of pure-imaginary type. In this particular case, the operator is analogous to the Compose_From_Cartesian function; it performs no arithmetic. 44.a

- The real (resp., imaginary) component of the result of a binary subtraction operator that yields a result of complex type is exact when its right operand is of pure-imaginary (resp., real) type. 45
- The real component of the result of the Conjugate function for the complex type is exact. 46
- When the point in the complex plane represented by the parameter X lies on the nonnegative real axis, the Argument function yields a result of zero. 47

Discussion: Argument(X + i*Y) is analogous to *EF*.Arctan(Y, X), where *EF* is an appropriate instance of Numerics.Generic_Elementary_Functions, except when X and Y are both zero, in which case the former yields the value zero while the latter raises Numerics.Argument_Error. 47.a

48 • When the value of the parameter Modulus is zero, the Compose_From_Polar function yields a result of zero.

49 • When the value of the parameter Argument is equal to a multiple of the quarter cycle, the result of the Compose_From_Polar function with specified cycle lies on one of the axes. In this case, one of its components is zero, and the other has the magnitude of the parameter Modulus.

50 • Exponentiation by a zero exponent yields the value one. Exponentiation by a unit exponent yields the value of the left operand. Exponentiation of the value one yields the value one. Exponentiation of the value zero yields the value zero, provided that the exponent is nonzero. When the left operand is of pure-imaginary type, one component of the result of the exponentiation operator is zero.

51 When the result, or a result component, of any operator of Numerics.Generic_Complex_Types has a mathematical definition in terms of a single arithmetic or relational operation, that result or result component exhibits the accuracy of the corresponding operation of the type Real.

52 Other accuracy requirements for the Modulus, Argument, and Compose_From_Polar functions, and accuracy requirements for the multiplication of a pair of complex operands or for division by a complex operand, all of which apply only in the strict mode, are given in G.2.6.

53 The sign of a zero result or zero result component yielded by a complex arithmetic operation or function is implementation defined when Real'Signed_Zeros is True.

53.a **Implementation defined:** The sign of a zero result (or a component thereof) from any operator or function in Numerics.Generic_Complex_Types, when Real'Signed_Zeros is True.

Implementation Permissions

54/5 {*AI12-0444-1*} The nongeneric equivalent packages can be actual instantiations of the generic package for the appropriate predefined type, though that is not required.

55/2 {*8652/0091*} {*AI95-00434-01*} Implementations may obtain the result of exponentiation of a complex or pure-imaginary operand by repeated complex multiplication, with arbitrary association of the factors and with a possible final complex reciprocation (when the exponent is negative). Implementations are also permitted to obtain the result of exponentiation of a complex operand, but not of a pure-imaginary operand, by converting the left operand to a polar representation; exponentiating the modulus by the given exponent; multiplying the argument by the given exponent; and reconverting to a Cartesian representation. Because of this implementation freedom, no accuracy requirement is imposed on complex exponentiation (except for the prescribed results given above, which apply regardless of the implementation method chosen).

Implementation Advice

56/5 {*AI12-0437-1*} Because the usual mathematical meaning of multiplication of a complex operand and a real operand is that of the scaling of both components of the former by the latter, an implementation should not perform this operation by first promoting the real operand to complex type and then performing a full complex multiplication. In systems that, in the future, support an Ada binding to ISO/IEC 60559:2020, the latter technique will not generate the required result when one of the components of the complex operand is infinite. (Explicit multiplication of the infinite component by the zero component obtained during promotion yields a NaN that propagates into the final result.) Analogous advice applies in the case of multiplication of a complex operand and a pure-imaginary operand, and in the case of division of a complex operand by a real or pure-imaginary operand.

56.a/2 **Implementation Advice:** Mixed real and complex operations (as well as pure-imaginary and complex operations) should not be performed by converting the real (resp. pure-imaginary) operand to complex.

{*AI12-0437-1*} Likewise, because the usual mathematical meaning of addition of a complex operand and a real operand is that the imaginary operand remains unchanged, an implementation should not perform this operation by first promoting the real operand to complex type and then performing a full complex addition. In implementations in which the Signed_Zeros attribute of the component type is True (and which therefore conform to ISO/IEC 60559:2020 in regard to the handling of the sign of zero in predefined arithmetic operations), the latter technique will not generate the required result when the imaginary component of the complex operand is a negatively signed zero. (Explicit addition of the negative zero to the zero obtained during promotion yields a positive zero.) Analogous advice applies in the case of addition of a complex operand and a pure-imaginary operand, and in the case of subtraction of a complex operand and a real or pure-imaginary operand. 57/5

Implementations in which Real'Signed_Zeros is True should attempt to provide a rational treatment of the signs of zero results and result components. As one example, the result of the Argument function should have the sign of the imaginary component of the parameter X when the point represented by that parameter lies on the positive real axis; as another, the sign of the imaginary component of the Compose_From_Polar function should be the same as (resp., the opposite of) that of the Argument parameter when that parameter has a value of zero and the Modulus parameter has a nonnegative (resp., negative) value. 58

Implementation Advice: If Real'Signed_Zeros is True for Numerics.Generic_Complex_Types, a rational treatment of the signs of zero results and result components should be provided. 58.a.1/3

Wording Changes from Ada 83

The semantics of Numerics.Generic_Complex_Types differs from Generic_Complex_Types as defined in ISO/IEC CD 13813 (for Ada 83) in the following ways: 58.a

- The generic package is a child of the package defining the Argument_Error exception. 58.b
- The nongeneric equivalents export types and constants with the same names as those exported by the generic package, rather than with names unique to the package. 58.c
- Implementations are not allowed to impose an optional restriction that the generic actual parameter associated with Real be unconstrained. (In view of the ability to declare variables of subtype Real'Base in implementations of Numerics.Generic_Complex_Types, this flexibility is no longer needed.) 58.d
- The dependence of the Argument function on the sign of a zero parameter component is tied to the value of Real'Signed_Zeros. 58.e
- Conformance to accuracy requirements is conditional. 58.f

Extensions to Ada 95

{*AI95-00161-01*} **Amendment Correction:** Added a pragma Preelaborable_Initialization to type Imaginary, so that it can be used in preelaborated units. 58.g/2

Wording Changes from Ada 95

{*8652/0020*} {*AI95-00126-01*} **Corrigendum:** Explicitly stated that the nongeneric equivalents of Generic_Complex_Types are pure. 58.h/2

G.1.2 Complex Elementary Functions

Static Semantics

The generic library package Numerics.Generic_Complex_Elementary_Functions has the following declaration: 1

2/5
```
{AI95-00434-01} {AI12-0241-1} with Ada.Numerics.Generic_Complex_Types;
generic
   with package Complex_Types is
         new Ada.Numerics.Generic_Complex_Types (<>);
   use Complex_Types;
package Ada.Numerics.Generic_Complex_Elementary_Functions
   with Pure, Nonblocking is
```

3
```
   function Sqrt (X : Complex)   return Complex;
   function Log  (X : Complex)   return Complex;
   function Exp  (X : Complex)   return Complex;
   function Exp  (X : Imaginary) return Complex;
   function "**" (Left : Complex;   Right : Complex)   return Complex;
   function "**" (Left : Complex;   Right : Real'Base) return Complex;
   function "**" (Left : Real'Base; Right : Complex)   return Complex;
```

4
```
   function Sin (X : Complex) return Complex;
   function Cos (X : Complex) return Complex;
   function Tan (X : Complex) return Complex;
   function Cot (X : Complex) return Complex;
```

5
```
   function Arcsin (X : Complex) return Complex;
   function Arccos (X : Complex) return Complex;
   function Arctan (X : Complex) return Complex;
   function Arccot (X : Complex) return Complex;
```

6
```
   function Sinh (X : Complex) return Complex;
   function Cosh (X : Complex) return Complex;
   function Tanh (X : Complex) return Complex;
   function Coth (X : Complex) return Complex;
```

7
```
   function Arcsinh (X : Complex) return Complex;
   function Arccosh (X : Complex) return Complex;
   function Arctanh (X : Complex) return Complex;
   function Arccoth (X : Complex) return Complex;
```

8
```
end Ada.Numerics.Generic_Complex_Elementary_Functions;
```

9/1 {*8652/0020*} {*AI95-00126-01*} The library package Numerics.Complex_Elementary_Functions is declared pure and defines the same subprograms as Numerics.Generic_Complex_Elementary_Functions, except that the predefined type Float is systematically substituted for Real'Base, and the Complex and Imaginary types exported by Numerics.Complex_Types are systematically substituted for Complex and Imaginary, throughout. Nongeneric equivalents of Numerics.Generic_Complex_Elementary_Functions corresponding to each of the other predefined floating point types are defined similarly, with the names Numerics.Short_Complex_Elementary_Functions, Numerics.Long_Complex_Elementary_Functions, etc.

9.a **Reason:** The nongeneric equivalents are provided to allow the programmer to construct simple mathematical applications without being required to understand and use generics.

10 The overloading of the Exp function for the pure-imaginary type is provided to give the user an alternate way to compose a complex value from a given modulus and argument. In addition to Compose_From_Polar(Rho, Theta) (see G.1.1), the programmer may write Rho * Exp(i * Theta).

11 The imaginary (resp., real) component of the parameter X of the forward hyperbolic (resp., trigonometric) functions and of the Exp function (and the parameter X, itself, in the case of the overloading of the Exp function for the pure-imaginary type) represents an angle measured in radians, as does the imaginary (resp., real) component of the result of the Log and inverse hyperbolic (resp., trigonometric) functions.

12 The functions have their usual mathematical meanings. However, the arbitrariness inherent in the placement of branch cuts, across which some of the complex elementary functions exhibit discontinuities, is eliminated by the following conventions:

13 • The imaginary component of the result of the Sqrt and Log functions is discontinuous as the parameter X crosses the negative real axis.

- The result of the exponentiation operator when the left operand is of complex type is discontinuous as that operand crosses the negative real axis. 14
- {*AI95-00185-01*} The imaginary component of the result of the Arcsin, Arccos, and Arctanh functions is discontinuous as the parameter X crosses the real axis to the left of –1.0 or the right of 1.0. 15/2
- {*AI95-00185-01*} The real component of the result of the Arctan and Arcsinh functions is discontinuous as the parameter X crosses the imaginary axis below –*i* or above *i*. 16/2
- {*AI95-00185-01*} The real component of the result of the Arccot function is discontinuous as the parameter X crosses the imaginary axis below –*i* or above *i*. 17/2
- The imaginary component of the Arccosh function is discontinuous as the parameter X crosses the real axis to the left of 1.0. 18
- The imaginary component of the result of the Arccoth function is discontinuous as the parameter X crosses the real axis between –1.0 and 1.0. 19

Discussion: {*AI95-00185-01*} The branch cuts come from the fact that the functions in question are really multi-valued in the complex domain, and that we have to pick one *principal value* to be the result of the function. Evidently we have much freedom in choosing where the branch cuts lie. However, we are adhering to the following principles which seem to lead to the more *natural* definitions: 19.a/2

- A branch cut should not intersect the real axis at a place where the corresponding real function is well-defined (in other words, the complex function should be an extension of the corresponding real function). 19.b/2
- Because all the functions in question are analytic, to ensure power series validity for the principal value, the branch cuts should be invariant by complex conjugation. 19.c/2
- For odd functions, to ensure that the principal value remains an odd function, the branch cuts should be invariant by reflection in the origin. 19.d/2

{*AI95-00185-01*} The computed results of the mathematically multivalued functions are rendered single-valued by the following conventions, which are meant to imply that the principal branch is an analytic continuation of the corresponding real-valued function in Numerics.Generic_Elementary_Functions. (For Arctan and Arccot, the single-argument function in question is that obtained from the two-argument version by fixing the second argument to be its default value.) 20/2

- The real component of the result of the Sqrt and Arccosh functions is nonnegative. 21
- The same convention applies to the imaginary component of the result of the Log function as applies to the result of the natural-cycle version of the Argument function of Numerics.Generic_Complex_Types (see G.1.1). 22
- The range of the real (resp., imaginary) component of the result of the Arcsin and Arctan (resp., Arcsinh and Arctanh) functions is approximately –π/2.0 to π/2.0. 23
- The real (resp., imaginary) component of the result of the Arccos and Arccot (resp., Arccoth) functions ranges from 0.0 to approximately π. 24
- The range of the imaginary component of the result of the Arccosh function is approximately –π to π. 25

In addition, the exponentiation operator inherits the single-valuedness of the Log function. 26

Dynamic Semantics

The exception Numerics.Argument_Error is raised by the exponentiation operator, signaling a parameter value outside the domain of the corresponding mathematical function, when the value of the left operand is zero and the real component of the exponent (or the exponent itself, when it is of real type) is zero. 27

The exception Constraint_Error is raised, signaling a pole of the mathematical function (analogous to dividing by zero), in the following cases, provided that Complex_Types.Real'Machine_Overflows is True: 28

29 • by the Log, Cot, and Coth functions, when the value of the parameter X is zero;

30 • by the exponentiation operator, when the value of the left operand is zero and the real component of the exponent (or the exponent itself, when it is of real type) is negative;

31 • by the Arctan and Arccot functions, when the value of the parameter X is ± *i*;

32 • by the Arctanh and Arccoth functions, when the value of the parameter X is ± 1.0.

33 [Constraint_Error can also be raised when a finite result overflows (see G.2.6); this may occur for parameter values sufficiently *near* poles, and, in the case of some of the functions, for parameter values having components of sufficiently large magnitude.] When Complex_Types.Real'Machine_Overflows is False, the result at poles is unspecified.

33.a **Reason:** The purpose of raising Constraint_Error (rather than Numerics.Argument_Error) at the poles of a function, when Float_Type'Machine_Overflows is True, is to provide continuous behavior as the actual parameters of the function approach the pole and finally reach it.

33.b/5 **Discussion:** {*AI12-0437-1*} It is anticipated that an Ada binding to ISO/IEC 60559:2020 will be developed in the future. As part of such a binding, the Machine_Overflows attribute of a conformant floating point type will be specified to yield False, which will permit implementations of the complex elementary functions to deliver results with an infinite component (and set the overflow flag defined by the binding) instead of raising Constraint_Error in overflow situations, when traps are disabled. Similarly, it is appropriate for the complex elementary functions to deliver results with an infinite component (and set the zero-divide flag defined by the binding) instead of raising Constraint_Error at poles, when traps are disabled. Finally, such a binding should also specify the behavior of the complex elementary functions, when sensible, given parameters with infinite components.

Implementation Requirements

34 In the implementation of Numerics.Generic_Complex_Elementary_Functions, the range of intermediate values allowed during the calculation of a final result shall not be affected by any range constraint of the subtype Complex_Types.Real.

34.a **Implementation Note:** Implementations of Numerics.Generic_Complex_Elementary_Functions written in Ada should therefore avoid declaring local variables of subtype Complex_Types.Real; the subtype Complex_Types.Real'Base should be used instead.

35 In the following cases, evaluation of a complex elementary function shall yield the *prescribed result* (or a result having the prescribed component), provided that the preceding rules do not call for an exception to be raised:

36 • When the parameter X has the value zero, the Sqrt, Sin, Arcsin, Tan, Arctan, Sinh, Arcsinh, Tanh, and Arctanh functions yield a result of zero; the Exp, Cos, and Cosh functions yield a result of one; the Arccos and Arccot functions yield a real result; and the Arccoth function yields an imaginary result.

37 • When the parameter X has the value one, the Sqrt function yields a result of one; the Log, Arccos, and Arccosh functions yield a result of zero; and the Arcsin function yields a real result.

38 • When the parameter X has the value –1.0, the Sqrt function yields the result

39 – *i* (resp., –*i*), when the sign of the imaginary component of X is positive (resp., negative), if Complex_Types.Real'Signed_Zeros is True;

40 – *i*, if Complex_Types.Real'Signed_Zeros is False;

41/2 • {*AI95-00434-01*} When the parameter X has the value –1.0, the Log function yields an imaginary result; and the Arcsin and Arccos functions yield a real result.

42 • When the parameter X has the value ± *i*, the Log function yields an imaginary result.

43 • Exponentiation by a zero exponent yields the value one. Exponentiation by a unit exponent yields the value of the left operand (as a complex value). Exponentiation of the value one yields the value one. Exponentiation of the value zero yields the value zero.

Discussion: It is possible to give many other prescribed results restricting the result to the real or imaginary axis when the parameter X is appropriately restricted to easily testable portions of the domain. We follow the proposed ISO/IEC standard for Generic_Complex_Elementary_Functions (for Ada 83), CD 13813, in not doing so, however. 43.a

Other accuracy requirements for the complex elementary functions, which apply only in the strict mode, are given in G.2.6. 44

The sign of a zero result or zero result component yielded by a complex elementary function is implementation defined when Complex_Types.Real'Signed_Zeros is True. 45

Implementation defined: The sign of a zero result (or a component thereof) from any operator or function in Numerics.Generic_Complex_Elementary_Functions, when Complex_Types.Real'Signed_Zeros is True. 45.a

Implementation Permissions

{*AI12-0444-1*} The nongeneric equivalent packages can be actual instantiations of the generic package with the appropriate predefined nongeneric equivalent of Numerics.Generic_Complex_Types, though that is not required; if they are, then the latter shall have been obtained by actual instantiation of Numerics.Generic_Complex_Types. 46/5

The exponentiation operator may be implemented in terms of the Exp and Log functions. Because this implementation yields poor accuracy in some parts of the domain, no accuracy requirement is imposed on complex exponentiation. 47

The implementation of the Exp function of a complex parameter X is allowed to raise the exception Constraint_Error, signaling overflow, when the real component of X exceeds an unspecified threshold that is approximately log(Complex_Types.Real'Safe_Last). This permission recognizes the impracticality of avoiding overflow in the marginal case that the exponential of the real component of X exceeds the safe range of Complex_Types.Real but both components of the final result do not. Similarly, the Sin and Cos (resp., Sinh and Cosh) functions are allowed to raise the exception Constraint_Error, signaling overflow, when the absolute value of the imaginary (resp., real) component of the parameter X exceeds an unspecified threshold that is approximately log(Complex_Types.Real'Safe_Last) + log(2.0). This permission recognizes the impracticality of avoiding overflow in the marginal case that the hyperbolic sine or cosine of the imaginary (resp., real) component of X exceeds the safe range of Complex_Types.Real but both components of the final result do not. 48

Implementation Advice

Implementations in which Complex_Types.Real'Signed_Zeros is True should attempt to provide a rational treatment of the signs of zero results and result components. For example, many of the complex elementary functions have components that are odd functions of one of the parameter components; in these cases, the result component should have the sign of the parameter component at the origin. Other complex elementary functions have zero components whose sign is opposite that of a parameter component at the origin, or is always positive or always negative. 49

Implementation Advice: If Complex_Types.Real'Signed_Zeros is True for Numerics.Generic_Complex_Elementary_Functions, a rational treatment of the signs of zero results and result components should be provided. 49.a.1/3

Wording Changes from Ada 83

The semantics of Numerics.Generic_Complex_Elementary_Functions differs from Generic_Complex_Elementary_Functions as defined in ISO/IEC CD 13814 (for Ada 83) in the following ways: 49.a

- The generic package is a child unit of the package defining the Argument_Error exception. 49.b
- The proposed Generic_Complex_Elementary_Functions standard (for Ada 83) specified names for the nongeneric equivalents, if provided. Here, those nongeneric equivalents are required. 49.c
- The generic package imports an instance of Numerics.Generic_Complex_Types rather than a long list of individual types and operations exported by such an instance. 49.d

49.e
- The dependence of the imaginary component of the Sqrt and Log functions on the sign of a zero parameter component is tied to the value of Complex_Types.Real'Signed_Zeros.

49.f
- Conformance to accuracy requirements is conditional.

Wording Changes from Ada 95

49.g/2 {*8652/0020*} {*AI95-00126-01*} **Corrigendum:** Explicitly stated that the nongeneric equivalents of Generic_Complex_Elementary_Functions are pure.

49.h/2 {*AI95-00185-01*} Corrected various inconsistencies in the definition of the branch cuts.

G.1.3 Complex Input-Output

1 The generic package Text_IO.Complex_IO defines procedures for the formatted input and output of complex values. The generic actual parameter in an instantiation of Text_IO.Complex_IO is an instance of Numerics.Generic_Complex_Types for some floating point subtype. Exceptional conditions are reported by raising the appropriate exception defined in Text_IO.

1.a **Implementation Note:** An implementation of Text_IO.Complex_IO can be built around an instance of Text_IO.Float_IO for the base subtype of Complex_Types.Real, where Complex_Types is the generic formal package parameter of Text_IO.Complex_IO. There is no need for an implementation of Text_IO.Complex_IO to parse real values.

Static Semantics

2 The generic library package Text_IO.Complex_IO has the following declaration:

2.a **Ramification:** Because this is a child of Text_IO, the declarations of the visible part of Text_IO are directly visible within it.

3/5
```
{AI12-0302-1} with Ada.Numerics.Generic_Complex_Types;
generic
   with package Complex_Types is
         new Ada.Numerics.Generic_Complex_Types (<>);
package Ada.Text_IO.Complex_IO
   with Global => in out synchronized is
```

4
```
   use Complex_Types;
```

5
```
   Default_Fore : Field := 2;
   Default_Aft  : Field := Real'Digits - 1;
   Default_Exp  : Field := 3;
```

6
```
   procedure Get (File  : in  File_Type;
                  Item  : out Complex;
                  Width : in  Field := 0);
   procedure Get (Item  : out Complex;
                  Width : in  Field := 0);
```

7
```
   procedure Put (File : in File_Type;
                  Item : in Complex;
                  Fore : in Field := Default_Fore;
                  Aft  : in Field := Default_Aft;
                  Exp  : in Field := Default_Exp);
   procedure Put (Item : in Complex;
                  Fore : in Field := Default_Fore;
                  Aft  : in Field := Default_Aft;
                  Exp  : in Field := Default_Exp);
```

8/5
```
{AI12-0241-1}    procedure Get (From : in  String;
                  Item : out Complex;
                  Last : out Positive)
      with Nonblocking;
   procedure Put (To   : out String;
                  Item : in  Complex;
                  Aft  : in  Field := Default_Aft;
                  Exp  : in  Field := Default_Exp)
      with Nonblocking;
```

```
end Ada.Text_IO.Complex_IO;
```

9

{*AI95-00328-01*} The library package Complex_Text_IO defines the same subprograms as Text_IO.Complex_IO, except that the predefined type Float is systematically substituted for Real, and the type Numerics.Complex_Types.Complex is systematically substituted for Complex throughout. Nongeneric equivalents of Text_IO.Complex_IO corresponding to each of the other predefined floating point types are defined similarly, with the names Short_Complex_Text_IO, Long_Complex_Text_IO, etc.

9.1/2

Reason: The nongeneric equivalents are provided to allow the programmer to construct simple mathematical applications without being required to understand and use generics.

9.a/2

The semantics of the Get and Put procedures are as follows:

10

```
procedure Get (File  : in  File_Type;
               Item  : out Complex;
               Width : in  Field := 0);
procedure Get (Item  : out Complex;
               Width : in  Field := 0);
```

11

{*8652/0092*} {*AI95-00029-01*} The input sequence is a pair of optionally signed real literals representing the real and imaginary components of a complex value. These components have the format defined for the corresponding Get procedure of an instance of Text_IO.Float_IO (see A.10.9) for the base subtype of Complex_Types.Real. The pair of components may be separated by a comma or surrounded by a pair of parentheses or both. Blanks are freely allowed before each of the components and before the parentheses and comma, if either is used. If the value of the parameter Width is zero, then

12/1

- line and page terminators are also allowed in these places;

13

- the components shall be separated by at least one blank or line terminator if the comma is omitted; and

14

- reading stops when the right parenthesis has been read, if the input sequence includes a left parenthesis, or when the imaginary component has been read, otherwise.

15

If a nonzero value of Width is supplied, then

15.1

- the components shall be separated by at least one blank if the comma is omitted; and

16

- exactly Width characters are read, or the characters (possibly none) up to a line terminator, whichever comes first (blanks are included in the count).

17

Reason: The parenthesized and comma-separated form is the form produced by Put on output (see below), and also by list-directed output in Fortran. The other allowed forms match several common styles of edit-directed output in Fortran, allowing most preexisting Fortran data files containing complex data to be read easily. When such files contain complex values with no separation between the real and imaginary components, the user will have to read those components separately, using an instance of Text_IO.Float_IO.

17.a

Returns, in the parameter Item, the value of type Complex that corresponds to the input sequence.

18

The exception Text_IO.Data_Error is raised if the input sequence does not have the required syntax or if the components of the complex value obtained are not of the base subtype of Complex_Types.Real.

19

```
procedure Put (File : in File_Type;
               Item : in Complex;
               Fore : in Field := Default_Fore;
               Aft  : in Field := Default_Aft;
               Exp  : in Field := Default_Exp);
procedure Put (Item : in Complex;
               Fore : in Field := Default_Fore;
               Aft  : in Field := Default_Aft;
               Exp  : in Field := Default_Exp);
```

20

21 Outputs the value of the parameter Item as a pair of decimal literals representing the real and imaginary components of the complex value, using the syntax of an aggregate. More specifically,

22 • outputs a left parenthesis;

23 • outputs the value of the real component of the parameter Item with the format defined by the corresponding Put procedure of an instance of Text_IO.Float_IO for the base subtype of Complex_Types.Real, using the given values of Fore, Aft, and Exp;

24 • outputs a comma;

25 • outputs the value of the imaginary component of the parameter Item with the format defined by the corresponding Put procedure of an instance of Text_IO.Float_IO for the base subtype of Complex_Types.Real, using the given values of Fore, Aft, and Exp;

26 • outputs a right parenthesis.

26.a **Discussion:** If the file has a bounded line length, a line terminator may be output implicitly before any element of the sequence itemized above.

26.b **Discussion:** The option of outputting the complex value as a pair of reals without additional punctuation is not provided, since it can be accomplished by outputting the real and imaginary components of the complex value separately.

27
```
procedure Get (From : in  String;
               Item : out Complex;
               Last : out Positive);
```

28/2 {*AI95-00434-01*} Reads a complex value from the beginning of the given string, following the same rule as the Get procedure that reads a complex value from a file, but treating the end of the string as a file terminator. Returns, in the parameter Item, the value of type Complex that corresponds to the input sequence. Returns in Last the index value such that From(Last) is the last character read.

29 The exception Text_IO.Data_Error is raised if the input sequence does not have the required syntax or if the components of the complex value obtained are not of the base subtype of Complex_Types.Real.

30
```
procedure Put (To   : out String;
               Item : in  Complex;
               Aft  : in  Field := Default_Aft;
               Exp  : in  Field := Default_Exp);
```

31 Outputs the value of the parameter Item to the given string as a pair of decimal literals representing the real and imaginary components of the complex value, using the syntax of an aggregate. More specifically,

32 • a left parenthesis, the real component, and a comma are left justified in the given string, with the real component having the format defined by the Put procedure (for output to a file) of an instance of Text_IO.Float_IO for the base subtype of Complex_Types.Real, using a value of zero for Fore and the given values of Aft and Exp;

33 • the imaginary component and a right parenthesis are right justified in the given string, with the imaginary component having the format defined by the Put procedure (for output to a file) of an instance of Text_IO.Float_IO for the base subtype of Complex_Types.Real, using a value for Fore that completely fills the remainder of the string, together with the given values of Aft and Exp.

33.a **Reason:** This rule is the one proposed in LSN-1051. Other rules were considered, including one that would have read "Outputs the value of the parameter Item to the given string, following the same rule as for output to a file, using a value for Fore such that the sequence of characters output exactly fills, or comes closest to filling, the string; in the latter case, the string is filled by inserting one extra blank immediately after the comma." While this latter rule might be considered the closest analogue to the rule for output to a string in Text_IO.Float_IO, it requires a more difficult and

inefficient implementation involving special cases when the integer part of one component is substantially longer than that of the other and the string is too short to allow both to be preceded by blanks. Unless such a special case applies, the latter rule might produce better columnar output if several such strings are ultimately output to a file, but very nearly the same output can be produced by outputting to the file directly, with the appropriate value of Fore; in any case, it might validly be assumed that output to a string is intended for further computation rather than for display, so that the precise formatting of the string to achieve a particular appearance is not the major concern.

The exception Text_IO.Layout_Error is raised if the given string is too short to hold the formatted output. 34

Implementation Permissions

Other exceptions declared (by renaming) in Text_IO may be raised by the preceding procedures in the appropriate circumstances, as for the corresponding procedures of Text_IO.Float_IO. 35

Extensions to Ada 95

{*AI95-00328-01*} Nongeneric equivalents for Text_IO.Complex_IO are added, to be consistent with all other language-defined Numerics generic packages. 35.a/2

Wording Changes from Ada 95

{*8652/0092*} {*AI95-00029-01*} **Corrigendum:** Clarified that the syntax of values read by Complex_IO is the same as that read by Text_IO.Float_IO. 35.b/2

G.1.4 The Package Wide_Text_IO.Complex_IO

Static Semantics

Implementations shall also provide the generic library package Wide_Text_IO.Complex_IO. Its declaration is obtained from that of Text_IO.Complex_IO by systematically replacing Text_IO by Wide_Text_IO and String by Wide_String; the description of its behavior is obtained by additionally replacing references to particular characters (commas, parentheses, etc.) by those for the corresponding wide characters. 1

G.1.5 The Package Wide_Wide_Text_IO.Complex_IO

Static Semantics

{*AI95-00285-01*} Implementations shall also provide the generic library package Wide_Wide_Text_IO.Complex_IO. Its declaration is obtained from that of Text_IO.Complex_IO by systematically replacing Text_IO by Wide_Wide_Text_IO and String by Wide_Wide_String; the description of its behavior is obtained by additionally replacing references to particular characters (commas, parentheses, etc.) by those for the corresponding wide wide characters. 1/2

Extensions to Ada 95

{*AI95-00285-01*} Package Wide_Wide_Text_IO.Complex_IO is new. (At least it wasn't called Incredibly_Wide_Text_IO.Complex_IO; maybe next time.) 1.a/2

G.2 Numeric Performance Requirements

Implementation Requirements

Implementations shall provide a user-selectable mode in which the accuracy and other numeric performance requirements detailed in the following subclauses are observed. This mode, referred to as the *strict mode*, may or may not be the default mode; it directly affects the results of the predefined arithmetic operations of real types and the results of the subprograms in children of the Numerics package, and 1

indirectly affects the operations in other language defined packages. Implementations shall also provide the opposing mode, which is known as the *relaxed mode*.

1.a **Reason:** On the assumption that the users of an implementation that does not support the Numerics Annex have no particular need for numerical performance, such an implementation has no obligation to meet any particular requirements in this area. On the other hand, users of an implementation that does support the Numerics Annex are provided with a way of ensuring that their programs achieve a known level of numerical performance and that the performance is portable to other such implementations. The relaxed mode is provided to allow implementers to offer an efficient but not fully accurate alternative in the case that the strict mode entails a time overhead that some users may find excessive. In some of its areas of impact, the relaxed mode may be fully equivalent to the strict mode.

1.b **Implementation Note:** The relaxed mode may, for example, be used to exploit the implementation of (some of) the elementary functions in hardware, when available. Such implementations often do not meet the accuracy requirements of the strict mode, or do not meet them over the specified range of parameter values, but compensate in other ways that may be important to the user, such as their extreme speed.

1.c **Ramification:** For implementations supporting the Numerics Annex, the choice of mode has no effect on the selection of a representation for a real type or on the values of attributes of a real type.

Implementation Permissions

2 Either mode may be the default mode.

2.a **Implementation defined:** Whether the strict mode or the relaxed mode is the default.

3/5 {*AI12-0444-1*} The two modes can be one and the same.

Extensions to Ada 83

3.a The choice between strict and relaxed numeric performance was not available in Ada 83.

G.2.1 Model of Floating Point Arithmetic

1 In the strict mode, the predefined operations of a floating point type shall satisfy the accuracy requirements specified here and shall avoid or signal overflow in the situations described. This behavior is presented in terms of a model of floating point arithmetic that builds on the concept of the canonical form (see A.5.3).

Static Semantics

2 Associated with each floating point type is an infinite set of model numbers. The model numbers of a type are used to define the accuracy requirements that have to be satisfied by certain predefined operations of the type; through certain attributes of the model numbers, they are also used to explain the meaning of a user-declared floating point type declaration. The model numbers of a derived type are those of the parent type; the model numbers of a subtype are those of its type.

3 The *model numbers* of a floating point type T are zero and all the values expressible in the canonical form (for the type T), in which *mantissa* has T'Model_Mantissa digits and *exponent* has a value greater than or equal to T'Model_Emin. (These attributes are defined in G.2.2.)

3.a **Discussion:** The model is capable of describing the behavior of most existing hardware that has a mantissa-exponent representation. As applied to a type T, it is parameterized by the values of T'Machine_Radix, T'Model_Mantissa, T'Model_Emin, T'Safe_First, and T'Safe_Last. The values of these attributes are determined by how, and how well, the hardware behaves. They in turn determine the set of model numbers and the safe range of the type, which figure in the accuracy and range (overflow avoidance) requirements.

3.b/5 {*AI12-0453-1*} In hardware that is free of arithmetic anomalies, T'Model_Mantissa, T'Model_Emin, T'Safe_First, and T'Safe_Last will yield the same values as T'Machine_Mantissa, T'Machine_Emin, T'Base'First, and T'Base'Last, respectively, and the model numbers in the safe range of the type T will coincide with the machine numbers of the type T. In less perfect hardware, it is not possible for the model-oriented attributes to have these optimal values, since the hardware, by definition, and therefore the implementation, cannot conform to the stringencies of the resulting model; in this case, the values yielded by the model-oriented parameters have to be made more conservative (that is, have to be penalized), with the result that the model numbers are more widely separated than the machine numbers, and the

safe range is a subrange of the base range. The implementation will then be able to conform to the requirements of the weaker model defined by the sparser set of model numbers and the smaller safe range.

A *model interval* of a floating point type is any interval whose bounds are model numbers of the type. The *model interval* of a type T *associated with a value v* is the smallest model interval of T that includes *v*. (The model interval associated with a model number of a type consists of that number only.) 4

Implementation Requirements

The accuracy requirements for the evaluation of certain predefined operations of floating point types are as follows. 5

Discussion: This subclause does not cover the accuracy of an operation of a static expression; such operations have to be evaluated exactly (see 4.9). It also does not cover the accuracy of the predefined attributes of a floating point subtype that yield a value of the type; such operations also yield exact results (see 3.5.8 and A.5.3). 5.a

An *operand interval* is the model interval, of the type specified for the operand of an operation, associated with the value of the operand. 6

For any predefined arithmetic operation that yields a result of a floating point type T, the required bounds on the result are given by a model interval of T (called the *result interval*) defined in terms of the operand values as follows: 7

- The result interval is the smallest model interval of T that includes the minimum and the maximum of all the values obtained by applying the (exact) mathematical operation to values arbitrarily selected from the respective operand intervals. 8

The result interval of an exponentiation is obtained by applying the above rule to the sequence of multiplications defined by the exponent, assuming arbitrary association of the factors, and to the final division in the case of a negative exponent. 9

The result interval of a conversion of a numeric value to a floating point type T is the model interval of T associated with the operand value, except when the source expression is of a fixed point type with a *small* that is not a power of T'Machine_Radix or is a fixed point multiplication or division either of whose operands has a *small* that is not a power of T'Machine_Radix; in these cases, the result interval is implementation defined. 10

Implementation defined: The result interval in certain cases of fixed-to-float conversion. 10.a

For any of the foregoing operations, the implementation shall deliver a value that belongs to the result interval when both bounds of the result interval are in the safe range of the result type T, as determined by the values of T'Safe_First and T'Safe_Last; otherwise, 11

- if T'Machine_Overflows is True, the implementation shall either deliver a value that belongs to the result interval or raise Constraint_Error; 12
- if T'Machine_Overflows is False, the result is implementation defined. 13

Implementation defined: The result of a floating point arithmetic operation in overflow situations, when the Machine_Overflows attribute of the result type is False. 13.a

{*AI12-0453-1*} For any predefined relation on operands of a floating point type T, the implementation may deliver any value (that is, either True or False) obtained by applying the (exact) mathematical comparison to values arbitrarily chosen from the respective operand intervals. 14/5

The result of a membership test is defined in terms of comparisons of the operand value with the lower and upper bounds of the given range or type mark (the usual rules apply to these comparisons). 15

Implementation Permissions

If the underlying floating point hardware implements division as multiplication by a reciprocal, the result interval for division (and exponentiation by a negative exponent) is implementation defined. 16

16.a **Implementation defined:** The result interval for division (or exponentiation by a negative exponent), when the floating point hardware implements division as multiplication by a reciprocal.

Wording Changes from Ada 83

16.b The Ada 95 model numbers of a floating point type that are in the safe range of the type are comparable to the Ada 83 safe numbers of the type. There is no analog of the Ada 83 model numbers. The Ada 95 model numbers, when not restricted to the safe range, are an infinite set.

Inconsistencies With Ada 83

16.c Giving the model numbers the hardware radix, instead of always a radix of two, allows (in conjunction with other changes) some borderline declared types to be represented with less precision than in Ada 83 (that is, with single precision, whereas Ada 83 would have used double precision). Because the lower precision satisfies the requirements of the model (and did so in Ada 83 as well), this change is viewed as a desirable correction of an anomaly, rather than a worrisome inconsistency. (Of course, the wider representation chosen in Ada 83 also remains eligible for selection in Ada 95.)

16.d/5 {*AI12-0453-1*} As an example of this phenomenon, assume that Float is represented in single precision and that a double precision type is also available. Also assume hexadecimal hardware with clean properties, for example certain IBM hardware. Then,

16.e
```
type T is digits Float'Digits range -Float'Last .. Float'Last;
```

16.f results in T being represented in double precision in Ada 83 and in single precision in Ada 95. The latter is intuitively correct; the former is counterintuitive. The reason why the double precision type is used in Ada 83 is that Float has model and safe numbers (in Ada 83) with 21 binary digits in their mantissas, as is required to model the hypothesized hexadecimal hardware using a binary radix; thus Float'Last, which is not a model number, is slightly outside the range of safe numbers of the single precision type, making that type ineligible for selection as the representation of T even though it provides adequate precision. In Ada 95, Float'Last (the same value as before) is a model number and is in the safe range of Float on the hypothesized hardware, making Float eligible for the representation of T.

Extensions to Ada 83

16.g Giving the model numbers the hardware radix allows for practical implementations on decimal hardware.

Wording Changes from Ada 83

16.h The wording of the model of floating point arithmetic has been simplified to a large extent.

G.2.2 Model-Oriented Attributes of Floating Point Types

1 In implementations that support the Numerics Annex, the model-oriented attributes of floating point types shall yield the values defined here, in both the strict and the relaxed modes. These definitions add conditions to those in A.5.3.

Static Semantics

2 For every subtype S of a floating point type *T*:

3/2 {*AI95-00256-01*} S'Model_Mantissa

Yields the number of digits in the mantissa of the canonical form of the model numbers of *T* (see A.5.3). The value of this attribute shall be greater than or equal to

3.1/2 $\lceil d \cdot \log(10) / \log(T\text{'Machine\_Radix}) \rceil + g$

3.2/2 where *d* is the requested decimal precision of *T*, and *g* is 0 if *T*'Machine_Radix is a positive power of 10 and 1 otherwise. In addition, *T*'Model_Mantissa shall be less than or equal to the value of *T*'Machine_Mantissa. This attribute yields a value of the type *universal_integer*.

3.a **Ramification:** S'Model_Epsilon, which is defined in terms of S'Model_Mantissa (see A.5.3), yields the absolute value of the difference between one and the next model number of the type *T* above one. It is equal to or larger than the absolute value of the difference between one and the next machine number of the type *T* above one.

S'Model_Emin 4

Yields the minimum exponent of the canonical form of the model numbers of *T* (see A.5.3). The value of this attribute shall be greater than or equal to the value of *T*'Machine_Emin. This attribute yields a value of the type *universal_integer*.

Ramification: S'Model_Small, which is defined in terms of S'Model_Emin (see A.5.3), yields the smallest positive (nonzero) model number of the type *T*. 4.a

S'Safe_First 5

Yields the lower bound of the safe range of *T*. The value of this attribute shall be a model number of *T* and greater than or equal to the lower bound of the base range of *T*. In addition, if *T* is declared by a floating_point_definition or is derived from such a type, and the floating_point_definition includes a real_range_specification specifying a lower bound of *lb*, then the value of this attribute shall be less than or equal to *lb*; otherwise, it shall be less than or equal to $-10.0^{4 \cdot d}$, where *d* is the requested decimal precision of *T*. This attribute yields a value of the type *universal_real*.

S'Safe_Last 6

Yields the upper bound of the safe range of *T*. The value of this attribute shall be a model number of *T* and less than or equal to the upper bound of the base range of *T*. In addition, if *T* is declared by a floating_point_definition or is derived from such a type, and the floating_point_definition includes a real_range_specification specifying an upper bound of *ub*, then the value of this attribute shall be greater than or equal to *ub*; otherwise, it shall be greater than or equal to $10.0^{4 \cdot d}$, where d is the requested decimal precision of *T*. This attribute yields a value of the type *universal_real*.

S'Model Denotes a function (of a parameter *X*) whose specification is given in A.5.3. If *X* is a model number of *T*, the function yields *X*; otherwise, it yields the value obtained by rounding or truncating *X* to either one of the adjacent model numbers of *T*. Constraint_Error is raised if the resulting model number is outside the safe range of S. A zero result has the sign of *X* when S'Signed_Zeros is True. 7

Subject to the constraints given above, the values of S'Model_Mantissa and S'Safe_Last are to be maximized, and the values of S'Model_Emin and S'Safe_First minimized, by the implementation as follows: 8

- First, S'Model_Mantissa is set to the largest value for which values of S'Model_Emin, S'Safe_First, and S'Safe_Last can be chosen so that the implementation satisfies the strict-mode requirements of G.2.1 in terms of the model numbers and safe range induced by these attributes. 9
- Next, S'Model_Emin is set to the smallest value for which values of S'Safe_First and S'Safe_Last can be chosen so that the implementation satisfies the strict-mode requirements of G.2.1 in terms of the model numbers and safe range induced by these attributes and the previously determined value of S'Model_Mantissa. 10
- {*AI05-0092-1*} Finally, S'Safe_First and S'Safe_Last are set (in either order) to the smallest and largest values, respectively, for which the implementation satisfies the strict-mode requirements of G.2.1 in terms of the model numbers and safe range induced by these attributes and the previously determined values of S'Model_Mantissa and S'Model_Emin. 11/3

Ramification: {*AI12-0437-1*} The following table shows appropriate attribute values for IEEE basic single and double precision types (ANSI/IEEE Std 754-1985, ISO/IEC 60559:2020). Here, we use the names IEEE_Float_32 and IEEE_Float_64, the names that would typically be declared in package Interfaces, in an implementation that supports IEEE arithmetic. In such an implementation, the attributes would typically be the same for Standard.Float and Long_Float, respectively. 11.a/5

Attribute	IEEE_Float_32	IEEE_Float_64	11.b

11.c

'Machine_Radix	2	2
'Machine_Mantissa	24	53
'Machine_Emin	-125	-1021
'Machine_Emax	128	1024
'Denorm	True	True
'Machine_Rounds	True	True
'Machine_Overflows	True/False	True/False
'Signed_Zeros	should be True	should be True

11.d

'Model_Mantissa	(same as 'Machine_Mantissa)	(same as 'Machine_Mantissa)
'Model_Emin	(same as 'Machine_Emin)	(same as 'Machine_Emin)
'Model_Epsilon	2.0**(-23)	2.0**(-52)
'Model_Small	2.0**(-126)	2.0**(-1022)
'Safe_First	-2.0**128*(1.0-2.0**(-24))	-2.0**1024*(1.0-2.0**(-53))
'Safe_Last	2.0**128*(1.0-2.0**(-24))	2.0**1024*(1.0-2.0**(-53))

11.e

'Digits	6	15
'Base'Digits	(same as 'Digits)	(same as 'Digits)

11.f

'First	(same as 'Safe_First)	(same as 'Safe_First)
'Last	(same as 'Safe_Last)	(same as 'Safe_Last)
'Size	32	64

11.g Note: 'Machine_Overflows can be True or False, depending on whether the Ada implementation raises Constraint_Error or delivers a signed infinity in overflow and zerodivide situations (and at poles of the elementary functions).

Wording Changes from Ada 95

11.h/2 {*AI95-00256-01*} Corrected the definition of Model_Mantissa to match that given in 3.5.8.

G.2.3 Model of Fixed Point Arithmetic

1 In the strict mode, the predefined arithmetic operations of a fixed point type shall satisfy the accuracy requirements specified here and shall avoid or signal overflow in the situations described.

Implementation Requirements

2 The accuracy requirements for the predefined fixed point arithmetic operations and conversions, and the results of relations on fixed point operands, are given below.

2.a **Discussion:** This subclause does not cover the accuracy of an operation of a static expression; such operations have to be evaluated exactly (see 4.9).

3 The operands of the fixed point adding operators, absolute value, and comparisons have the same type. These operations are required to yield exact results, unless they overflow.

4 Multiplications and divisions are allowed between operands of any two fixed point types; the result has to be (implicitly or explicitly) converted to some other numeric type. For purposes of defining the accuracy rules, the multiplication or division and the conversion are treated as a single operation whose accuracy depends on three types (those of the operands and the result). For decimal fixed point types, the attribute T'Round may be used to imply explicit conversion with rounding (see 3.5.10).

5 When the result type is a floating point type, the accuracy is as given in G.2.1. For some combinations of the operand and result types in the remaining cases, the result is required to belong to a small set of values called the *perfect result set*; for other combinations, it is required merely to belong to a generally larger and implementation-defined set of values called the *close result set*. When the result type is a decimal fixed point type, the perfect result set contains a single value; thus, operations on decimal types are always fully specified.

5.a **Implementation defined:** The definition of *close result set*, which determines the accuracy of certain fixed point multiplications and divisions.

6/5 {*AI12-0453-1*} When one operand of a fixed-fixed multiplication or division is of type *universal_real*, that operand is not implicitly converted in the usual sense, since the context does not determine a unique target type, but the accuracy of the result of the multiplication or division (that is, whether the result has

to belong to the perfect result set or merely the close result set) depends on the value of the operand of type *universal_real* and on the types of the other operand and of the result.

Discussion: {*AI12-0453-1*} We need not consider here the multiplication or division of two such operands, since in that case either the operation is evaluated exactly (that is, it is an operation of a static expression all of whose operators are of a root numeric type) or it is considered to be an operation of a floating point type. 6.a/5

For a fixed point multiplication or division whose (exact) mathematical result is *v*, and for the conversion of a value *v* to a fixed point type, the perfect result set and close result set are defined as follows: 7

- If the result type is an ordinary fixed point type with a *small* of *s*, 8
  - if *v* is an integer multiple of *s*, then the perfect result set contains only the value *v*; 9
  - otherwise, it contains the integer multiple of *s* just below *v* and the integer multiple of *s* just above *v*. 10

  The close result set is an implementation-defined set of consecutive integer multiples of *s* containing the perfect result set as a subset. 11
- If the result type is a decimal type with a *small* of *s*, 12
  - if *v* is an integer multiple of *s*, then the perfect result set contains only the value *v*; 13
  - {*AI05-0264-1*} otherwise, if truncation applies, then it contains only the integer multiple of *s* in the direction toward zero, whereas if rounding applies, then it contains only the nearest integer multiple of *s* (with ties broken by rounding away from zero). 14/3

  The close result set is an implementation-defined set of consecutive integer multiples of *s* containing the perfect result set as a subset. 15

  Ramification: As a consequence of subsequent rules, this case does not arise when the operand types are also decimal types. 15.a
- If the result type is an integer type, 16
  - if *v* is an integer, then the perfect result set contains only the value *v*; 17
  - otherwise, it contains the integer nearest to the value *v* (if *v* lies equally distant from two consecutive integers, the perfect result set contains the one that is further from zero). 18

  The close result set is an implementation-defined set of consecutive integers containing the perfect result set as a subset. 19

{*AI12-0453-1*} The result of a fixed point multiplication or division shall belong either to the perfect result set or to the close result set, as described below, if overflow does not occur. In the following cases, if the result type is a fixed point type, let *s* be its *small*; otherwise, that is when the result type is an integer type, let *s* be 1.0. 20/5

- {*AI12-0453-1*} For a multiplication or division neither of whose operands is of type *universal_real*, let *l* and *r* be the *smalls* of the left and right operands. For a multiplication, if $(l \cdot r) / s$ is an integer or the reciprocal of an integer (the *smalls* are said to be "compatible" in this case), the result shall belong to the perfect result set; otherwise, it belongs to the close result set. For a division, if $l / (r \cdot s)$ is an integer or the reciprocal of an integer (that is, the *smalls* are compatible), the result shall belong to the perfect result set; otherwise, it belongs to the close result set. 21/5

  Ramification: When the operand and result types are all decimal types, their *smalls* are necessarily compatible; the same is true when they are all ordinary fixed point types with binary *smalls*. 21.a
- For a multiplication or division having one *universal_real* operand with a value of *v*, note that it is always possible to factor *v* as an integer multiple of a "compatible" *small*, but the integer multiple may be "too big". If there exists a factorization in which that multiple is less than some implementation-defined limit, the result shall belong to the perfect result set; otherwise, it belongs to the close result set. 22

22.a **Implementation defined:** Conditions on a *universal_real* operand of a fixed point multiplication or division for which the result shall be in the *perfect result set*.

23/5 {*AI12-0300-1*} A multiplication P * Q of an operand of a fixed point type F by an operand of type Integer, or vice versa, and a division P / Q of an operand of a fixed point type F by an operand of type Integer, are also allowed. In these cases, the result has the type of F; explicit conversion of the result is never required. The accuracy required in these cases is the same as that required for a multiplication F(P * Q) or a division F(P / Q) obtained by interpreting the operand of the integer type to have a fixed point type with a *small* of 1.0.

24 The accuracy of the result of a conversion from an integer or fixed point type to a fixed point type, or from a fixed point type to an integer type, is the same as that of a fixed point multiplication of the source value by a fixed point operand having a *small* of 1.0 and a value of 1.0, as given by the foregoing rules. The result of a conversion from a floating point type to a fixed point type shall belong to the close result set. The result of a conversion of a *universal_real* operand to a fixed point type shall belong to the perfect result set.

25 The possibility of overflow in the result of a predefined arithmetic operation or conversion yielding a result of a fixed point type T is analogous to that for floating point types, except for being related to the base range instead of the safe range. If all of the permitted results belong to the base range of T, then the implementation shall deliver one of the permitted results; otherwise,

26 • if T'Machine_Overflows is True, the implementation shall either deliver one of the permitted results or raise Constraint_Error;

27 • if T'Machine_Overflows is False, the result is implementation defined.

27.a **Implementation defined:** The result of a fixed point arithmetic operation in overflow situations, when the Machine_Overflows attribute of the result type is False.

Inconsistencies With Ada 83

27.b Since the values of a fixed point type are now just the integer multiples of its *small*, the possibility of using extra bits available in the chosen representation for extra accuracy rather than for increasing the base range would appear to be removed, raising the possibility that some fixed point expressions will yield less accurate results than in Ada 83. However, this is partially offset by the ability of an implementation to choose a smaller default *small* than before. Of course, if it does so for a type T then T'Small will have a different value than it previously had.

27.c The accuracy requirements in the case of incompatible *smalls* are relaxed to foster wider support for nonbinary *smalls*. If this relaxation is exploited for a type that was previously supported, lower accuracy could result; however, there is no particular incentive to exploit the relaxation in such a case.

Wording Changes from Ada 83

27.d The fixed point accuracy requirements are now expressed without reference to model or safe numbers, largely because the full generality of the former model was never exploited in the case of fixed point types (particularly in regard to operand perturbation). Although the new formulation in terms of perfect result sets and close result sets is still verbose, it can be seen to distill down to two cases:

27.e • a case where the result must be the exact result, if the exact result is representable, or, if not, then either one of the adjacent values of the type (in some subcases only one of those adjacent values is allowed);

27.f • a case where the accuracy is not specified by the language.

Wording Changes from Ada 2012

27.g/5 {*AI12-0300-1*} **Correction:** Reworded the fixed*integer accuracy requirements to clarify that the only allowed integer type in such operations is Standard.Integer. We make this correction as readers of the Reference Manual have been confused on this point.

G.2.4 Accuracy Requirements for the Elementary Functions

In the strict mode, the performance of Numerics.Generic_Elementary_Functions shall be as specified here. 1

Implementation Requirements

When an exception is not raised, the result of evaluating a function in an instance *EF* of Numerics.Generic_Elementary_Functions belongs to a *result interval*, defined as the smallest model interval of *EF*.Float_Type that contains all the values of the form $f \cdot (1.0 + d)$, where f is the exact value of the corresponding mathematical function at the given parameter values, d is a real number, and $|d|$ is less than or equal to the function's *maximum relative error*. The function delivers a value that belongs to the result interval when both of its bounds belong to the safe range of *EF*.Float_Type; otherwise, 2

- if *EF*.Float_Type'Machine_Overflows is True, the function either delivers a value that belongs to the result interval or raises Constraint_Error, signaling overflow; 3
- if *EF*.Float_Type'Machine_Overflows is False, the result is implementation defined. 4

  Implementation defined: The result of an elementary function reference in overflow situations, when the Machine_Overflows attribute of the result type is False. 4.a

The maximum relative error exhibited by each function is as follows: 5

- $2.0 \cdot$ *EF*.Float_Type'Model_Epsilon, in the case of the Sqrt, Sin, and Cos functions; 6
- $4.0 \cdot$ *EF*.Float_Type'Model_Epsilon, in the case of the Log, Exp, Tan, Cot, and inverse trigonometric functions; and 7
- $8.0 \cdot$ *EF*.Float_Type'Model_Epsilon, in the case of the forward and inverse hyperbolic functions. 8

The maximum relative error exhibited by the exponentiation operator, which depends on the values of the operands, is $(4.0 + |\text{Right} \cdot \log(\text{Left})| / 32.0) \cdot$ *EF*.Float_Type'Model_Epsilon. 9

The maximum relative error given above applies throughout the domain of the forward trigonometric functions when the Cycle parameter is specified. When the Cycle parameter is omitted, the maximum relative error given above applies only when the absolute value of the angle parameter X is less than or equal to some implementation-defined *angle threshold*, which shall be at least *EF*.Float_Type'Machine_Radix $^{\lfloor EF.\text{Float\_Type'Machine\_Mantissa}/2 \rfloor}$. Beyond the angle threshold, the accuracy of the forward trigonometric functions is implementation defined. 10

Implementation defined: The value of the *angle threshold*, within which certain elementary functions, complex arithmetic operations, and complex elementary functions yield results conforming to a maximum relative error bound. 10.a

Implementation defined: The accuracy of certain elementary functions for parameters beyond the angle threshold. 10.b

Implementation Note: The angle threshold indirectly determines the amount of precision that the implementation has to maintain during argument reduction. 10.c

{*AI95-00434-01*} {*AI12-0451-1*} The prescribed results specified in A.5.1 for certain functions at particular parameter values take precedence over the maximum relative error bounds; effectively, they narrow to a single value the result interval allowed by the maximum relative error bounds. Additional rules with a similar effect are given by Table G.1 for the inverse trigonometric functions, at particular parameter values for which the mathematical result is possibly not a model number of *EF*.Float_Type (or is, indeed, even transcendental). In each table entry, the values of the parameters are such that the result lies on the axis between two quadrants; the corresponding accuracy rule, which takes precedence over the maximum relative error bounds, is that the result interval is the model interval of *EF*.Float_Type associated with the exact mathematical result given in the table. 11/5

This paragraph was deleted. 12/1

13 The last line of the table is meant to apply when *EF*.Float_Type'Signed_Zeros is False; the two lines just above it, when *EF*.Float_Type'Signed_Zeros is True and the parameter Y has a zero value with the indicated sign.

Table G.1: Tightly Approximated Elementary Function Results

Function	Value of X	Value of Y	Exact Result when Cycle Specified	Exact Result when Cycle Omitted
Arcsin	1.0	n.a.	Cycle/4.0	π/2.0
Arcsin	–1.0	n.a.	–Cycle/4.0	–π/2.0
Arccos	0.0	n.a.	Cycle/4.0	π/2.0
Arccos	–1.0	n.a.	Cycle/2.0	π
Arctan and Arccot	0.0	positive	Cycle/4.0	π/2.0
Arctan and Arccot	0.0	negative	–Cycle/4.0	–π/2.0
Arctan and Arccot	negative	+0.0	Cycle/2.0	π
Arctan and Arccot	negative	–0.0	–Cycle/2.0	–π
Arctan and Arccot	negative	0.0	Cycle/2.0	π

14 The amount by which the result of an inverse trigonometric function is allowed to spill over into a quadrant adjacent to the one corresponding to the principal branch, as given in A.5.1, is limited. The rule is that the result belongs to the smallest model interval of *EF*.Float_Type that contains both boundaries of the quadrant corresponding to the principal branch. This rule also takes precedence over the maximum relative error bounds, effectively narrowing the result interval allowed by them.

15 Finally, the following specifications also take precedence over the maximum relative error bounds:

16 • The absolute value of the result of the Sin, Cos, and Tanh functions never exceeds one.

17 • The absolute value of the result of the Coth function is never less than one.

18 • The result of the Cosh function is never less than one.

Implementation Advice

19 The versions of the forward trigonometric functions without a Cycle parameter should not be implemented by calling the corresponding version with a Cycle parameter of 2.0*Numerics.Pi, since this will not provide the required accuracy in some portions of the domain. For the same reason, the version of Log without a Base parameter should not be implemented by calling the corresponding version with a Base parameter of Numerics.e.

19.a.1/2 **Implementation Advice:** For elementary functions, the forward trigonometric functions without a Cycle parameter should not be implemented by calling the corresponding version with a Cycle parameter. Log without a Base parameter should not be implemented by calling Log with a Base parameter.

Wording Changes from Ada 83

19.a The semantics of Numerics.Generic_Elementary_Functions differs from Generic_Elementary_Functions as defined in ISO/IEC DIS 11430 (for Ada 83) in the following ways related to the accuracy specified for strict mode:

19.b • The maximum relative error bounds use the Model_Epsilon attribute instead of the Base'Epsilon attribute.

- The accuracy requirements are expressed in terms of result intervals that are model intervals. On the one hand, this facilitates the description of the required results in the presence of underflow; on the other hand, it slightly relaxes the requirements expressed in ISO/IEC DIS 11430. 19.c

G.2.5 Performance Requirements for Random Number Generation

In the strict mode, the performance of Numerics.Float_Random and Numerics.Discrete_Random shall be as specified here. 1

Implementation Requirements

Two different calls to the time-dependent Reset procedure shall reset the generator to different states, provided that the calls are separated in time by at least one second and not more than fifty years. 2

The implementation's representations of generator states and its algorithms for generating random numbers shall yield a period of at least $2^{31}-2$; much longer periods are desirable but not required. 3

{*AI12-0453-1*} The implementations of Numerics.Float_Random.Random and Numerics.Discrete_Random.Random shall pass at least 85% of the individual trials in a suite of statistical tests. For Numerics.Float_Random, the tests are applied directly to the floating point values generated (that is, they are not converted to integers first), while for Numerics.Discrete_Random they are applied to the generated values of various discrete types. Each test suite performs 6 different tests, with each test repeated 10 times, yielding a total of 60 individual trials. An individual trial is deemed to pass if the chi-square value (or other statistic) calculated for the observed counts or distribution falls within the range of values corresponding to the 2.5 and 97.5 percentage points for the relevant degrees of freedom (that is, it shall be neither too high nor too low). For the purpose of determining the degrees of freedom, measurement categories are combined whenever the expected counts are fewer than 5. 4/5

Implementation Note: In the floating point random number test suite, the generator is reset to a time-dependent state at the beginning of the run. The test suite incorporates the following tests, adapted from D. E. Knuth, *The Art of Computer Programming, vol. 2: Seminumerical Algorithms.* In the descriptions below, the given number of degrees of freedom is the number before reduction due to any necessary combination of measurement categories with small expected counts; it is one less than the number of measurement categories. 4.a

- Proportional Distribution Test (a variant of the Equidistribution Test). The interval 0.0 .. 1.0 is partitioned into K subintervals. K is chosen randomly between 4 and 25 for each repetition of the test, along with the boundaries of the subintervals (subject to the constraint that at least 2 of the subintervals have a width of 0.001 or more). 5000 random floating point numbers are generated. The counts of random numbers falling into each subinterval are tallied and compared with the expected counts, which are proportional to the widths of the subintervals. The number of degrees of freedom for the chi-square test is K–1. 4.b
- Gap Test. The bounds of a range A .. B, with $0.0 \le A < B \le 1.0$, are chosen randomly for each repetition of the test, subject to the constraint that $0.2 \le B-A \le 0.6$. Random floating point numbers are generated until 5000 falling into the range A .. B have been encountered. Each of these 5000 is preceded by a "gap" (of length greater than or equal to 0) of consecutive random numbers not falling into the range A .. B. The counts of gaps of each length from 0 to 15, and of all lengths greater than 15 lumped together, are tallied and compared with the expected counts. Let $P = B-A$. The probability that a gap has a length of L is $(1-P)^{L} \cdot P$ for $L \le 15$, while the probability that a gap has a length of 16 or more is $(1-P)^{16}$. The number of degrees of freedom for the chi-square test is 16. 4.c
- Permutation Test. 5000 tuples of 4 different random floating point numbers are generated. (An entire 4-tuple is discarded in the unlikely event that it contains any two exactly equal components.) The counts of each of the 4! = 24 possible relative orderings of the components of the 4-tuples are tallied and compared with the expected counts. Each of the possible relative orderings has an equal probability. The number of degrees of freedom for the chi-square test is 23. 4.d
- Increasing-Runs Test. Random floating point numbers are generated until 5000 increasing runs have been observed. An "increasing run" is a sequence of random numbers in strictly increasing order; it is followed by a random number that is strictly smaller than the preceding random number. (A run under construction is entirely discarded in the unlikely event that one random number is followed immediately by an exactly equal random number.) The decreasing random number that follows an increasing run is discarded and not included with the next increasing run. The counts of increasing runs of each length from 1 to 4, and of all lengths greater than 4 lumped together, are tallied and compared with the expected counts. The probability 4.e

that an increasing run has a length of L is $1/L! - 1/(L+1)!$ for $L \leq 4$, while the probability that an increasing run has a length of 5 or more is 1/5!. The number of degrees of freedom for the chi-square test is 4.

4.f
- Decreasing-Runs Test. The test is similar to the Increasing Runs Test, but with decreasing runs.

4.g
- Maximum-of-t Test (with $t = 5$). 5000 tuples of 5 random floating point numbers are generated. The maximum of the components of each 5-tuple is determined and raised to the 5th power. The uniformity of the resulting values over the range 0.0 .. 1.0 is tested as in the Proportional Distribution Test.

4.h
Implementation Note: In the discrete random number test suite, Numerics.Discrete_Random is instantiated as described below. The generator is reset to a time-dependent state after each instantiation. The test suite incorporates the following tests, adapted from D. E. Knuth (*op. cit.*) and other sources. The given number of degrees of freedom for the chi-square test is reduced by any necessary combination of measurement categories with small expected counts, as described above.

4.i
- Equidistribution Test. In each repetition of the test, a number R between 2 and 30 is chosen randomly, and Numerics.Discrete_Random is instantiated with an integer subtype whose range is 1 .. R. 5000 integers are generated randomly from this range. The counts of occurrences of each integer in the range are tallied and compared with the expected counts, which have equal probabilities. The number of degrees of freedom for the chi-square test is R–1.

4.j
- Simplified Poker Test. Numerics.Discrete_Random is instantiated once with an enumeration subtype representing the 13 denominations (Two through Ten, Jack, Queen, King, and Ace) of an infinite deck of playing cards. 2000 “poker” hands (5-tuples of values of this subtype) are generated randomly. The counts of hands containing exactly K different denominations ($1 \leq K \leq 5$) are tallied and compared with the expected counts. The probability that a hand contains exactly K different denominations is given by a formula in Knuth. The number of degrees of freedom for the chi-square test is 4.

4.k
- Coupon Collector's Test. Numerics.Discrete_Random is instantiated in each repetition of the test with an integer subtype whose range is 1 .. R, where R varies systematically from 2 to 11. Integers are generated randomly from this range until each value in the range has occurred, and the number K of integers generated is recorded. This constitutes a “coupon collector's segment” of length K. 2000 such segments are generated. The counts of segments of each length from R to R+29, and of all lengths greater than R+29 lumped together, are tallied and compared with the expected counts. The probability that a segment has any given length is given by formulas in Knuth. The number of degrees of freedom for the chi-square test is 30.

4.l
- Craps Test (Lengths of Games). Numerics.Discrete_Random is instantiated once with an integer subtype whose range is 1 .. 6 (representing the six numbers on a die). 5000 craps games are played, and their lengths are recorded. (The length of a craps game is the number of rolls of the pair of dice required to produce a win or a loss. A game is won on the first roll if the dice show 7 or 11; it is lost if they show 2, 3, or 12. If the dice show some other sum on the first roll, it is called the *point*, and the game is won if and only if the point is rolled again before a 7 is rolled.) The counts of games of each length from 1 to 18, and of all lengths greater than 18 lumped together, are tallied and compared with the expected counts. For $2 \leq S \leq 12$, let D_S be the probability that a roll of a pair of dice shows the sum S, and let $Q_S(L) = D_S \cdot (1 - (D_S + D_7))^{L-2} \cdot (D_S + D_7)$. Then, the probability that a game has a length of 1 is $D_7 + D_{11} + D_2 + D_3 + D_{12}$ and, for $L > 1$, the probability that a game has a length of L is $Q_4(L) + Q_5(L) + Q_6(L) + Q_8(L) + Q_9(L) + Q_{10}(L)$. The number of degrees of freedom for the chi-square test is 18.

4.m
- Craps Test (Lengths of Passes). This test is similar to the last, but enough craps games are played for 3000 losses to occur. A string of wins followed by a loss is called a *pass*, and its length is the number of wins preceding the loss. The counts of passes of each length from 0 to 7, and of all lengths greater than 7 lumped together, are tallied and compared with the expected counts. For $L \geq 0$, the probability that a pass has a length of L is $W^L \cdot (1-W)$, where W, the probability that a game ends in a win, is 244.0/495.0. The number of degrees of freedom for the chi-square test is 8.

4.n
- Collision Test. Numerics.Discrete_Random is instantiated once with an integer or enumeration type representing binary bits. 15 successive calls on the Random function are used to obtain the bits of a 15-bit binary integer between 0 and 32767. 3000 such integers are generated, and the number of collisions (integers previously generated) is counted and compared with the expected count. A chi-square test is not used to assess the number of collisions; rather, the limits on the number of collisions, corresponding to the 2.5 and 97.5 percentage points, are (from formulas in Knuth) 112 and 154. The test passes if and only if the number of collisions is in this range.

G.2.6 Accuracy Requirements for Complex Arithmetic

1 In the strict mode, the performance of Numerics.Generic_Complex_Types and Numerics.Generic_-Complex_Elementary_Functions shall be as specified here.

Implementation Requirements

{*AI12-0453-1*} When an exception is not raised, the result of evaluating a real function of an instance *CT* of Numerics.Generic_Complex_Types (that is, a function that yields a value of subtype *CT*.Real'Base or *CT*.Imaginary) belongs to a result interval defined as for a real elementary function (see G.2.4). 2/5

{*AI12-0453-1*} When an exception is not raised, each component of the result of evaluating a complex function of such an instance, or of an instance of Numerics.Generic_Complex_Elementary_Functions obtained by instantiating the latter with *CT* (that is, a function that yields a value of subtype *CT*.Complex), also belongs to a *result interval*. The result intervals for the components of the result are either defined by a *maximum relative error* bound or by a *maximum box error* bound. When the result interval for the real (resp., imaginary) component is defined by maximum relative error, it is defined as for that of a real function, relative to the exact value of the real (resp., imaginary) part of the result of the corresponding mathematical function. When defined by maximum box error, the result interval for a component of the result is the smallest model interval of *CT*.Real that contains all the values of the corresponding part of $f \cdot (1.0 + d)$, where f is the exact complex value of the corresponding mathematical function at the given parameter values, d is complex, and $|d|$ is less than or equal to the given maximum box error. The function delivers a value that belongs to the result interval (or a value both of whose components belong to their respective result intervals) when both bounds of the result interval(s) belong to the safe range of *CT*.Real; otherwise, 3/5

Discussion: The maximum relative error could be specified separately for each component, but we do not take advantage of that freedom here. 3.a

Discussion: Note that $f \cdot (1.0 + d)$ defines a small circular region of the complex plane centered at f, and the result intervals for the real and imaginary components of the result define a small rectangular box containing that circle. 3.b

Reason: Box error is used when the computation of the result risks loss of significance in a component due to cancellation. 3.c

Ramification: The components of a complex function that exhibits bounded relative error in each component have to have the correct sign. In contrast, one of the components of a complex function that exhibits bounded box error may have the wrong sign, since the dimensions of the box containing the result are proportional to the modulus of the mathematical result and not to either component of the mathematical result individually. Thus, for example, the box containing the computed result of a complex function whose mathematical result has a large modulus but lies very close to the imaginary axis might well straddle that axis, allowing the real component of the computed result to have the wrong sign. In this case, the distance between the computed result and the mathematical result is, nevertheless, a small fraction of the modulus of the mathematical result. 3.d

- if *CT*.Real'Machine_Overflows is True, the function either delivers a value that belongs to the result interval (or a value both of whose components belong to their respective result intervals) or raises Constraint_Error, signaling overflow; 4
- if *CT*.Real'Machine_Overflows is False, the result is implementation defined. 5

Implementation defined: The result of a complex arithmetic operation or complex elementary function reference in overflow situations, when the Machine_Overflows attribute of the corresponding real type is False. 5.a

{*AI95-00434-01*} The error bounds for particular complex functions are tabulated in Table G.2. In the table, the error bound is given as the coefficient of *CT*.Real'Model_Epsilon. 6/2

This paragraph was deleted. 7/1

Table G.2: Error Bounds for Particular Complex Functions

Function or Operator	Nature of Result	Nature of Bound	Error Bound
Modulus	real	max. rel. error	3.0
Argument	real	max. rel. error	4.0

Compose_From_Polar	complex	max. rel. error	3.0
"*" (both operands complex)	complex	max. box error	5.0
"/" (right operand complex)	complex	max. box error	13.0
Sqrt	complex	max. rel. error	6.0
Log	complex	max. box error	13.0
Exp (complex parameter)	complex	max. rel. error	7.0
Exp (imaginary parameter)	complex	max. rel. error	2.0
Sin, Cos, Sinh, and Cosh	complex	max. rel. error	11.0
Tan, Cot, Tanh, and Coth	complex	max. rel. error	35.0
inverse trigonometric	complex	max. rel. error	14.0
inverse hyperbolic	complex	max. rel. error	14.0

8 The maximum relative error given above applies throughout the domain of the Compose_From_Polar function when the Cycle parameter is specified. When the Cycle parameter is omitted, the maximum relative error applies only when the absolute value of the parameter Argument is less than or equal to the angle threshold (see G.2.4). For the Exp function, and for the forward hyperbolic (resp., trigonometric) functions, the maximum relative error given above likewise applies only when the absolute value of the imaginary (resp., real) component of the parameter X (or the absolute value of the parameter itself, in the case of the Exp function with a parameter of pure-imaginary type) is less than or equal to the angle threshold. For larger angles, the accuracy is implementation defined.

8.a **Implementation defined:** The accuracy of certain complex arithmetic operations and certain complex elementary functions for parameters (or components thereof) beyond the angle threshold.

9 The prescribed results specified in G.1.2 for certain functions at particular parameter values take precedence over the error bounds; effectively, they narrow to a single value the result interval allowed by the error bounds for a component of the result. Additional rules with a similar effect are given below for certain inverse trigonometric and inverse hyperbolic functions, at particular parameter values for which a component of the mathematical result is transcendental. In each case, the accuracy rule, which takes precedence over the error bounds, is that the result interval for the stated result component is the model interval of *CT*.Real associated with the component's exact mathematical value. The cases in question are as follows:

10 • When the parameter X has the value zero, the real (resp., imaginary) component of the result of the Arccot (resp., Arccoth) function is in the model interval of *CT*.Real associated with the value π/2.0.

11 • When the parameter X has the value one, the real component of the result of the Arcsin function is in the model interval of *CT*.Real associated with the value π/2.0.

12 • When the parameter X has the value –1.0, the real component of the result of the Arcsin (resp., Arccos) function is in the model interval of *CT*.Real associated with the value –π/2.0 (resp., π).

12.a **Discussion:** It is possible to give many other prescribed results in which a component of the parameter is restricted to a similar model interval when the parameter X is appropriately restricted to an easily testable portion of the domain. We follow the proposed ISO/IEC standard for Generic_Complex_Elementary_Functions (for Ada 83) in not doing so, however.

13/2 {*AI95-00434-01*} The amount by which a component of the result of an inverse trigonometric or inverse hyperbolic function is allowed to spill over into a quadrant adjacent to the one corresponding to the principal branch, as given in G.1.2, is limited. The rule is that the result belongs to the smallest model

interval of *CT*.Real that contains both boundaries of the quadrant corresponding to the principal branch. This rule also takes precedence over the maximum error bounds, effectively narrowing the result interval allowed by them.

Finally, the results allowed by the error bounds are narrowed by one further rule: The absolute value of each component of the result of the Exp function, for a pure-imaginary parameter, never exceeds one. 14

Implementation Advice

The version of the Compose_From_Polar function without a Cycle parameter should not be implemented by calling the corresponding version with a Cycle parameter of 2.0*Numerics.Pi, since this will not provide the required accuracy in some portions of the domain. 15

Implementation Advice: For complex arithmetic, the Compose_From_Polar function without a Cycle parameter should not be implemented by calling Compose_From_Polar with a Cycle parameter. 15.a.1/2

Wording Changes from Ada 83

The semantics of Numerics.Generic_Complex_Types and Numerics.Generic_Complex_Elementary_Functions differs from Generic_Complex_Types and Generic_Complex_Elementary_Functions as defined in ISO/IEC CDs 13813 and 13814 (for Ada 83) in ways analogous to those identified for the elementary functions in G.2.4. In addition, we do not generally specify the signs of zero results (or result components), although those proposed standards do. 15.a

G.3 Vector and Matrix Manipulation

{*AI95-00296-01*} Types and operations for the manipulation of real vectors and matrices are provided in Generic_Real_Arrays, which is defined in G.3.1. Types and operations for the manipulation of complex vectors and matrices are provided in Generic_Complex_Arrays, which is defined in G.3.2. Both of these library units are generic children of the predefined package Numerics (see A.5). Nongeneric equivalents of these packages for each of the predefined floating point types are also provided as children of Numerics. 1/2

Discussion: Vector and matrix manipulation is defined in the Numerics Annex, rather than in the core, because it is considered to be a specialized need of (some) numeric applications. 1.a/2

These packages provide facilities that are similar to and replace those found in ISO/IEC 13813:1998 *Information technology — Programming languages — Generic packages of real and complex type declarations and basic operations for Ada (including vector and matrix types)*. (The other facilities provided by that Standard were already provided in Ada 95.) In addition to the main facilities of that Standard, these packages also include subprograms for the solution of linear equations, matrix inversion, determinants, and the determination of the eigenvalues and eigenvectors of real symmetric matrices and Hermitian matrices. 1.b/2

Extensions to Ada 95

{*AI95-00296-01*} {*AI05-0299-1*} This subclause It just provides an introduction to the following subclauses. 1.c/3

G.3.1 Real Vectors and Matrices

Static Semantics

{*AI95-00296-01*} {*AI95-00418-01*} The generic library package Numerics.Generic_Real_Arrays has the following declaration: 1/2

```
{AI12-0241-1} generic                                                    2/5
   type Real is digits <>;
package Ada.Numerics.Generic_Real_Arrays
   with Pure, Nonblocking is

   -- Types                                                             3/2

   type Real_Vector is array (Integer range <>) of Real'Base;           4/2
   type Real_Matrix is array (Integer range <>, Integer range <>)
                                                  of Real'Base;
```

5/2
```
   -- Subprograms for Real_Vector types
```
6/2
```
   -- Real_Vector arithmetic operations
```
7/2
```
   function "+"   (Right : Real_Vector)       return Real_Vector;
   function "-"   (Right : Real_Vector)       return Real_Vector;
   function "abs" (Right : Real_Vector)       return Real_Vector;
```
8/2
```
   function "+"   (Left, Right : Real_Vector) return Real_Vector;
   function "-"   (Left, Right : Real_Vector) return Real_Vector;
```
9/2
```
   function "*"   (Left, Right : Real_Vector) return Real'Base;
```
10/2
```
   function "abs" (Right : Real_Vector)       return Real'Base;
```
11/2
```
   -- Real_Vector scaling operations
```
12/2
```
   function "*" (Left : Real'Base;   Right : Real_Vector)
      return Real_Vector;
   function "*" (Left : Real_Vector; Right : Real'Base)
      return Real_Vector;
   function "/" (Left : Real_Vector; Right : Real'Base)
      return Real_Vector;
```
13/2
```
   -- Other Real_Vector operations
```
14/2
```
   function Unit_Vector (Index : Integer;
                         Order : Positive;
                         First : Integer := 1) return Real_Vector;
```
15/2
```
   -- Subprograms for Real_Matrix types
```
16/2
```
   -- Real_Matrix arithmetic operations
```
17/2
```
   function "+"       (Right : Real_Matrix) return Real_Matrix;
   function "-"       (Right : Real_Matrix) return Real_Matrix;
   function "abs"     (Right : Real_Matrix) return Real_Matrix;
   function Transpose (X     : Real_Matrix) return Real_Matrix;
```
18/2
```
   function "+" (Left, Right : Real_Matrix) return Real_Matrix;
   function "-" (Left, Right : Real_Matrix) return Real_Matrix;
   function "*" (Left, Right : Real_Matrix) return Real_Matrix;
```
19/2
```
   function "*" (Left, Right : Real_Vector) return Real_Matrix;
```
20/2
```
   function "*" (Left : Real_Vector; Right : Real_Matrix)
      return Real_Vector;
   function "*" (Left : Real_Matrix; Right : Real_Vector)
      return Real_Vector;
```
21/2
```
   -- Real_Matrix scaling operations
```
22/2
```
   function "*" (Left : Real'Base;   Right : Real_Matrix)
      return Real_Matrix;
   function "*" (Left : Real_Matrix; Right : Real'Base)
      return Real_Matrix;
   function "/" (Left : Real_Matrix; Right : Real'Base)
      return Real_Matrix;
```
23/2
```
   -- Real_Matrix inversion and related operations
```
24/2
```
   function Solve (A : Real_Matrix; X : Real_Vector) return Real_Vector;
   function Solve (A, X : Real_Matrix) return Real_Matrix;
   function Inverse (A : Real_Matrix) return Real_Matrix;
   function Determinant (A : Real_Matrix) return Real'Base;
```
25/2
```
   -- Eigenvalues and vectors of a real symmetric matrix
```
26/2
```
   function Eigenvalues (A : Real_Matrix) return Real_Vector;
```
27/2
```
   procedure Eigensystem (A       : in  Real_Matrix;
                          Values  : out Real_Vector;
                          Vectors : out Real_Matrix);
```
28/2
```
   -- Other Real_Matrix operations
```
29/2
```
   function Unit_Matrix (Order            : Positive;
                         First_1, First_2 : Integer := 1)
                                            return Real_Matrix;
```

```
end Ada.Numerics.Generic_Real_Arrays;
```

30/2

{*AI95-00296-01*} The library package Numerics.Real_Arrays is declared pure and defines the same types and subprograms as Numerics.Generic_Real_Arrays, except that the predefined type Float is systematically substituted for Real'Base throughout. Nongeneric equivalents for each of the other predefined floating point types are defined similarly, with the names Numerics.Short_Real_Arrays, Numerics.Long_Real_Arrays, etc.

31/2

Reason: The nongeneric equivalents are provided to allow the programmer to construct simple mathematical applications without being required to understand and use generics, and to be consistent with other Numerics packages.

31.a/2

{*AI95-00296-01*} Two types are defined and exported by Numerics.Generic_Real_Arrays. The composite type Real_Vector is provided to represent a vector with components of type Real; it is defined as an unconstrained, one-dimensional array with an index of type Integer. The composite type Real_Matrix is provided to represent a matrix with components of type Real; it is defined as an unconstrained, two-dimensional array with indices of type Integer.

32/2

{*AI95-00296-01*} The effect of the various subprograms is as described below. In most cases the subprograms are described in terms of corresponding scalar operations of the type Real; any exception raised by those operations is propagated by the array operation. Moreover, the accuracy of the result for each individual component is as defined for the scalar operation unless stated otherwise.

33/2

{*AI95-00296-01*} In the case of those operations which are defined to *involve an inner product*, Constraint_Error may be raised if an intermediate result is outside the range of Real'Base even though the mathematical final result would not be.

34/2

```
function "+"   (Right : Real_Vector) return Real_Vector;
function "-"   (Right : Real_Vector) return Real_Vector;
function "abs" (Right : Real_Vector) return Real_Vector;
```

35/2

{*AI95-00296-01*} Each operation returns the result of applying the corresponding operation of the type Real to each component of Right. The index range of the result is Right'Range.

36/2

```
function "+" (Left, Right : Real_Vector) return Real_Vector;
function "-" (Left, Right : Real_Vector) return Real_Vector;
```

37/2

{*AI95-00296-01*} Each operation returns the result of applying the corresponding operation of the type Real to each component of Left and the matching component of Right. The index range of the result is Left'Range. Constraint_Error is raised if Left'Length is not equal to Right'Length.

38/2

```
function "*" (Left, Right : Real_Vector) return Real'Base;
```

39/2

{*AI95-00296-01*} This operation returns the inner product of Left and Right. Constraint_Error is raised if Left'Length is not equal to Right'Length. This operation involves an inner product.

40/2

```
function "abs" (Right : Real_Vector) return Real'Base;
```

41/2

{*AI95-00418-01*} This operation returns the L2-norm of Right (the square root of the inner product of the vector with itself).

42/2

Discussion: Normalization of vectors is a frequent enough operation that it is useful to provide the norm as a basic operation. Furthermore, implementing the norm is not entirely straightforward, because the inner product might overflow while the final norm does not. An implementation cannot merely return Sqrt (X * X), it has to cope with a possible overflow of the inner product.

42.a/2

Implementation Note: While the definition is given in terms of an inner product, the norm doesn't "involve an inner product" in the technical sense. The reason is that it has accuracy requirements substantially different from those applicable to inner products; and that cancellations cannot occur, because all the terms are positive, so there is no possibility of intermediate overflow.

42.b/2

```
function "*" (Left : Real'Base; Right : Real_Vector) return Real_Vector;
```

43/2

{*AI95-00296-01*} This operation returns the result of multiplying each component of Right by the scalar Left using the "*" operation of the type Real. The index range of the result is Right'Range.

44/2

45/2
```
function "*" (Left : Real_Vector; Right : Real'Base) return Real_Vector;
function "/" (Left : Real_Vector; Right : Real'Base) return Real_Vector;
```

46/2 {*AI95-00296-01*} Each operation returns the result of applying the corresponding operation of the type Real to each component of Left and to the scalar Right. The index range of the result is Left'Range.

47/2
```
function Unit_Vector (Index : Integer;
                      Order : Positive;
                      First : Integer := 1) return Real_Vector;
```

48/2 {*AI95-00296-01*} This function returns a *unit vector* with Order components and a lower bound of First. All components are set to 0.0 except for the Index component which is set to 1.0. Constraint_Error is raised if Index < First, Index > First + Order – 1 or if First + Order – 1 > Integer'Last.

49/2
```
function "+"   (Right : Real_Matrix) return Real_Matrix;
function "-"   (Right : Real_Matrix) return Real_Matrix;
function "abs" (Right : Real_Matrix) return Real_Matrix;
```

50/2 {*AI95-00296-01*} Each operation returns the result of applying the corresponding operation of the type Real to each component of Right. The index ranges of the result are those of Right.

51/2
```
function Transpose (X : Real_Matrix) return Real_Matrix;
```

52/2 {*AI95-00296-01*} This function returns the transpose of a matrix X. The first and second index ranges of the result are X'Range(2) and X'Range(1) respectively.

53/2
```
function "+" (Left, Right : Real_Matrix) return Real_Matrix;
function "-" (Left, Right : Real_Matrix) return Real_Matrix;
```

54/2 {*AI95-00296-01*} Each operation returns the result of applying the corresponding operation of the type Real to each component of Left and the matching component of Right. The index ranges of the result are those of Left. Constraint_Error is raised if Left'Length(1) is not equal to Right'Length(1) or Left'Length(2) is not equal to Right'Length(2).

55/2
```
function "*" (Left, Right : Real_Matrix) return Real_Matrix;
```

56/2 {*AI95-00296-01*} This operation provides the standard mathematical operation for matrix multiplication. The first and second index ranges of the result are Left'Range(1) and Right'Range(2) respectively. Constraint_Error is raised if Left'Length(2) is not equal to Right'Length(1). This operation involves inner products.

57/2
```
function "*" (Left, Right : Real_Vector) return Real_Matrix;
```

58/2 {*AI95-00296-01*} This operation returns the outer product of a (column) vector Left by a (row) vector Right using the operation "*" of the type Real for computing the individual components. The first and second index ranges of the result are Left'Range and Right'Range respectively.

59/2
```
function "*" (Left : Real_Vector; Right : Real_Matrix) return Real_Vector;
```

60/2 {*AI95-00296-01*} This operation provides the standard mathematical operation for multiplication of a (row) vector Left by a matrix Right. The index range of the (row) vector result is Right'Range(2). Constraint_Error is raised if Left'Length is not equal to Right'Length(1). This operation involves inner products.

61/2
```
function "*" (Left : Real_Matrix; Right : Real_Vector) return Real_Vector;
```

62/2 {*AI95-00296-01*} This operation provides the standard mathematical operation for multiplication of a matrix Left by a (column) vector Right. The index range of the (column) vector result is Left'Range(1). Constraint_Error is raised if Left'Length(2) is not equal to Right'Length. This operation involves inner products.

```
function "*" (Left : Real'Base; Right : Real_Matrix) return Real_Matrix;
```
63/2

{*AI95-00296-01*} This operation returns the result of multiplying each component of Right by the scalar Left using the "*" operation of the type Real. The index ranges of the result are those of Right.
64/2

```
function "*" (Left : Real_Matrix; Right : Real'Base) return Real_Matrix;
function "/" (Left : Real_Matrix; Right : Real'Base) return Real_Matrix;
```
65/2

{*AI95-00296-01*} Each operation returns the result of applying the corresponding operation of the type Real to each component of Left and to the scalar Right. The index ranges of the result are those of Left.
66/2

```
function Solve (A : Real_Matrix; X : Real_Vector) return Real_Vector;
```
67/2

{*AI95-00296-01*} This function returns a vector Y such that X is (nearly) equal to A * Y. This is the standard mathematical operation for solving a single set of linear equations. The index range of the result is A'Range(2). Constraint_Error is raised if A'Length(1), A'Length(2), and X'Length are not equal. Constraint_Error is raised if the matrix A is ill-conditioned.
68/2

Discussion: The text says that Y is such that “X is (nearly) equal to A * Y” rather than “X is equal to A * Y” because rounding errors may mean that there is no value of Y such that X is exactly equal to A * Y. On the other hand it does not mean that any old rough value will do. The algorithm given under Implementation Advice should be followed.
68.a/2

The requirement to raise Constraint_Error if the matrix is ill-conditioned is really a reflection of what will happen if the matrix is ill-conditioned. See Implementation Advice. We do not make any attempt to define ill-conditioned formally.
68.b/2

These remarks apply to all versions of Solve and Inverse.
68.c/2

```
function Solve (A, X : Real_Matrix) return Real_Matrix;
```
69/2

{*AI95-00296-01*} This function returns a matrix Y such that X is (nearly) equal to A * Y. This is the standard mathematical operation for solving several sets of linear equations. The index ranges of the result are A'Range(2) and X'Range(2). Constraint_Error is raised if A'Length(1), A'Length(2), and X'Length(1) are not equal. Constraint_Error is raised if the matrix A is ill-conditioned.
70/2

```
function Inverse (A : Real_Matrix) return Real_Matrix;
```
71/2

{*AI95-00296-01*} This function returns a matrix B such that A * B is (nearly) equal to the unit matrix. The index ranges of the result are A'Range(2) and A'Range(1). Constraint_Error is raised if A'Length(1) is not equal to A'Length(2). Constraint_Error is raised if the matrix A is ill-conditioned.
72/2

```
function Determinant (A : Real_Matrix) return Real'Base;
```
73/2

{*AI95-00296-01*} This function returns the determinant of the matrix A. Constraint_Error is raised if A'Length(1) is not equal to A'Length(2).
74/2

```
function Eigenvalues(A : Real_Matrix) return Real_Vector;
```
75/2

{*AI95-00296-01*} This function returns the eigenvalues of the symmetric matrix A as a vector sorted into order with the largest first. Constraint_Error is raised if A'Length(1) is not equal to A'Length(2). The index range of the result is A'Range(1). Argument_Error is raised if the matrix A is not symmetric.
76/2

```
procedure Eigensystem(A       : in  Real_Matrix;
                      Values  : out Real_Vector;
                      Vectors : out Real_Matrix);
```
77/2

{*AI95-00296-01*} {*AI05-0047-1*} This procedure computes both the eigenvalues and eigenvectors of the symmetric matrix A. The out parameter Values is the same as that obtained by calling the function Eigenvalues. The out parameter Vectors is a matrix whose columns are the eigenvectors of the matrix A. The order of the columns corresponds to the order of the eigenvalues. The
78/3

eigenvectors are normalized and mutually orthogonal (they are orthonormal), including when there are repeated eigenvalues. Constraint_Error is raised if A'Length(1) is not equal to A'Length(2), or if Values'Range is not equal to A'Range(1), or if the index ranges of the parameter Vectors are not equal to those of A. Argument_Error is raised if the matrix A is not symmetric. Constraint_Error is also raised in implementation-defined circumstances if the algorithm used does not converge quickly enough.

78.a/3 **Ramification:** {*AI05-0047-1*} There is no requirement on the absolute direction of the returned eigenvectors. Thus they might be multiplied by -1. It is only the ratios of the components that matter. This is standard practice.

79/2
```
function Unit_Matrix (Order            : Positive;
                      First_1, First_2 : Integer := 1) return Real_Matrix;
```

80/2 {*AI95-00296-01*} This function returns a square *unit matrix* with Order**2 components and lower bounds of First_1 and First_2 (for the first and second index ranges respectively). All components are set to 0.0 except for the main diagonal, whose components are set to 1.0. Constraint_Error is raised if First_1 + Order – 1 > Integer'Last or First_2 + Order – 1 > Integer'Last.

Implementation Requirements

81/2 {*AI95-00296-01*} Accuracy requirements for the subprograms Solve, Inverse, Determinant, Eigenvalues and Eigensystem are implementation defined.

81.a/2 **Implementation defined:** The accuracy requirements for the subprograms Solve, Inverse, Determinant, Eigenvalues and Eigensystem for type Real_Matrix.

82/2 For operations not involving an inner product, the accuracy requirements are those of the corresponding operations of the type Real in both the strict mode and the relaxed mode (see G.2).

83/2 For operations involving an inner product, no requirements are specified in the relaxed mode. In the strict mode the modulus of the absolute error of the inner product *X***Y* shall not exceed *g***abs*(*X*)***abs*(*Y*) where *g* is defined as

84/2 *g* = *X*'Length * Real'Machine_Radix**(1 – Real'Model_Mantissa)

85/2 {*AI95-00418-01*} For the L2-norm, no accuracy requirements are specified in the relaxed mode. In the strict mode the relative error on the norm shall not exceed *g* / 2.0 + 3.0 * Real'Model_Epsilon where *g* is defined as above.

85.a/2 **Reason:** This is simply the combination of the error on the inner product with the error on Sqrt. A first order computation would lead to 2.0 * Real'Model_Epsilon above, but we are adding an extra Real'Model_Epsilon to account for higher order effects.

Documentation Requirements

86/2 {*AI95-00296-01*} Implementations shall document any techniques used to reduce cancellation errors such as extended precision arithmetic.

86.a/2 **Documentation Requirement:** Any techniques used to reduce cancellation errors in Numerics.Generic_Real_Arrays shall be documented.

86.b/2 **Implementation Note:** The above accuracy requirement is met by the canonical implementation of the inner product by multiplication and addition using the corresponding operations of type Real'Base and performing the cumulative addition using ascending indices. Note however, that some hardware provides special operations for the computation of the inner product and although these may be fast they may not meet the accuracy requirement specified. See Accuracy and Stability of Numerical Algorithms By N J Higham (ISBN 0-89871-355-2), Section 3.1.

86.c/3 {*AI05-0047-1*} Note moreover that the componentwise accuracy requirements are not met by subcubic methods for matrix multiplication such as that devised by Strassen. These methods, which are typically used for the fast multiplication of very large matrices (that is, order more than a few thousands), have normwise accuracy properties. If it is desired to use such methods, then distinct subprograms should be provided (perhaps in a child package). See Section 22.2.2 in the above reference.

Implementation Permissions

{*AI95-00296-01*} {*AI12-0444-1*} The nongeneric equivalent packages can be actual instantiations of the generic package for the appropriate predefined type, though that is not required. 87/5

Implementation Advice

{*AI95-00296-01*} {*AI05-0264-1*} Implementations should implement the Solve and Inverse functions using established techniques such as LU decomposition with row interchanges followed by back and forward substitution. Implementations are recommended to refine the result by performing an iteration on the residuals; if this is done, then it should be documented. 88/3

Implementation Advice: Solve and Inverse for Numerics.Generic_Real_Arrays should be implemented using established techniques such as LU decomposition and the result should be refined by an iteration on the residuals. 88.a/2

It is not the intention that any special provision should be made to determine whether a matrix is ill-conditioned or not. The naturally occurring overflow (including division by zero) which will result from executing these functions with an ill-conditioned matrix and thus raise Constraint_Error is sufficient. 89/2

Discussion: There isn't any advice for the implementation to document with this paragraph. 89.a/2

The test that a matrix is symmetric should be performed by using the equality operator to compare the relevant components. 90/2

Implementation Advice: The equality operator should be used to test that a matrix in Numerics.Generic_Real_Arrays is symmetric. 90.a/2

{*AI05-0047-1*} An implementation should minimize the circumstances under which the algorithm used for Eigenvalues and Eigensystem fails to converge. 91/3

Implementation Advice: An implementation should minimize the circumstances under which the algorithm used for Numerics.Generic_Real_Arrays.Eigenvalues and Numerics.Generic_Real_Arrays.Eigensystem fails to converge. 91.a.1/3

Implementation Note: J. H. Wilkinson is the acknowledged expert in this area. See for example Wilkinson, J. H., and Reinsch, C. , Linear Algebra , vol II of Handbook for Automatic Computation, Springer-Verlag, or Wilkinson, J. H., The Algebraic Eigenvalue Problem, Oxford University Press. 91.a/3

Extensions to Ada 95

{*AI95-00296-01*} The package Numerics.Generic_Real_Arrays and its nongeneric equivalents are new. 91.b/2

Wording Changes from Ada 2005

{*AI05-0047-1*} **Correction:** Corrected various accuracy and definition issues. 91.c/3

G.3.2 Complex Vectors and Matrices

Static Semantics

{*AI95-00296-01*} The generic library package Numerics.Generic_Complex_Arrays has the following declaration: 1/2

```
{AI12-0241-1} with Ada.Numerics.Generic_Real_Arrays,                    2/5
Ada.Numerics.Generic_Complex_Types;
generic
   with package Real_Arrays   is new
      Ada.Numerics.Generic_Real_Arrays   (<>);
   use Real_Arrays;
   with package Complex_Types is new
      Ada.Numerics.Generic_Complex_Types (Real);
   use Complex_Types;
package Ada.Numerics.Generic_Complex_Arrays
   with Pure, Nonblocking is

   -- Types                                                             3/2
```

```
4/2         type Complex_Vector is array (Integer range <>) of Complex;
            type Complex_Matrix is array (Integer range <>,
                                          Integer range <>) of Complex;

5/2         -- Subprograms for Complex_Vector types

6/2         -- Complex_Vector selection, conversion and composition operations

7/2         function Re (X : Complex_Vector) return Real_Vector;
            function Im (X : Complex_Vector) return Real_Vector;

8/2         procedure Set_Re (X  : in out Complex_Vector;
                              Re : in     Real_Vector);
            procedure Set_Im (X  : in out Complex_Vector;
                              Im : in     Real_Vector);

9/2         function Compose_From_Cartesian (Re     : Real_Vector)
               return Complex_Vector;
            function Compose_From_Cartesian (Re, Im : Real_Vector)
               return Complex_Vector;

10/2        function Modulus  (X     : Complex_Vector) return Real_Vector;
            function "abs"    (Right : Complex_Vector) return Real_Vector
                                                          renames Modulus;
            function Argument (X     : Complex_Vector) return Real_Vector;
            function Argument (X     : Complex_Vector;
                               Cycle : Real'Base)      return Real_Vector;

11/2        function Compose_From_Polar (Modulus, Argument : Real_Vector)
               return Complex_Vector;
            function Compose_From_Polar (Modulus, Argument : Real_Vector;
                                         Cycle             : Real'Base)
               return Complex_Vector;

12/2        -- Complex_Vector arithmetic operations

13/2        function "+"       (Right  : Complex_Vector) return Complex_Vector;
            function "-"       (Right  : Complex_Vector) return Complex_Vector;
            function Conjugate (X      : Complex_Vector) return Complex_Vector;

14/2        function "+"  (Left, Right : Complex_Vector) return Complex_Vector;
            function "-"  (Left, Right : Complex_Vector) return Complex_Vector;

15/2        function "*"  (Left, Right : Complex_Vector) return Complex;

16/3    {AI05-0047-1}   function "abs"     (Right : Complex_Vector) return Real'Base;

17/2        -- Mixed Real_Vector and Complex_Vector arithmetic operations

18/2        function "+" (Left  : Real_Vector;
                          Right : Complex_Vector) return Complex_Vector;
            function "+" (Left  : Complex_Vector;
                          Right : Real_Vector)    return Complex_Vector;
            function "-" (Left  : Real_Vector;
                          Right : Complex_Vector) return Complex_Vector;
            function "-" (Left  : Complex_Vector;
                          Right : Real_Vector)    return Complex_Vector;

19/2        function "*" (Left  : Real_Vector;    Right : Complex_Vector)
               return Complex;
            function "*" (Left  : Complex_Vector; Right : Real_Vector)
               return Complex;

20/2        -- Complex_Vector scaling operations

21/2        function "*" (Left  : Complex;
                          Right : Complex_Vector) return Complex_Vector;
            function "*" (Left  : Complex_Vector;
                          Right : Complex)        return Complex_Vector;
            function "/" (Left  : Complex_Vector;
                          Right : Complex)        return Complex_Vector;
```

```
function "*" (Left  : Real'Base;                                          22/2
              Right : Complex_Vector) return Complex_Vector;
function "*" (Left  : Complex_Vector;
              Right : Real'Base)      return Complex_Vector;
function "/" (Left  : Complex_Vector;
              Right : Real'Base)      return Complex_Vector;

-- Other Complex_Vector operations                                        23/2

function Unit_Vector (Index : Integer;                                    24/2
                      Order : Positive;
                      First : Integer := 1) return Complex_Vector;

-- Subprograms for Complex_Matrix types                                   25/2

-- Complex_Matrix selection, conversion and composition operations        26/2

function Re (X : Complex_Matrix) return Real_Matrix;                      27/2
function Im (X : Complex_Matrix) return Real_Matrix;

procedure Set_Re (X  : in out Complex_Matrix;                             28/2
                  Re : in     Real_Matrix);
procedure Set_Im (X  : in out Complex_Matrix;
                  Im : in     Real_Matrix);

function Compose_From_Cartesian (Re     : Real_Matrix)                    29/2
   return Complex_Matrix;
function Compose_From_Cartesian (Re, Im : Real_Matrix)
   return Complex_Matrix;

function Modulus  (X     : Complex_Matrix) return Real_Matrix;            30/2
function "abs"    (Right : Complex_Matrix) return Real_Matrix
                                              renames Modulus;

function Argument (X     : Complex_Matrix) return Real_Matrix;            31/2
function Argument (X     : Complex_Matrix;
                   Cycle : Real'Base)      return Real_Matrix;

function Compose_From_Polar (Modulus, Argument : Real_Matrix)             32/2
   return Complex_Matrix;
function Compose_From_Polar (Modulus, Argument : Real_Matrix;
                             Cycle             : Real'Base)
   return Complex_Matrix;

-- Complex_Matrix arithmetic operations                                   33/2

function "+"       (Right : Complex_Matrix) return Complex_Matrix;        34/2
function "-"       (Right : Complex_Matrix) return Complex_Matrix;
function Conjugate (X     : Complex_Matrix) return Complex_Matrix;
function Transpose (X     : Complex_Matrix) return Complex_Matrix;

function "+" (Left, Right : Complex_Matrix) return Complex_Matrix;        35/2
function "-" (Left, Right : Complex_Matrix) return Complex_Matrix;
function "*" (Left, Right : Complex_Matrix) return Complex_Matrix;

function "*" (Left, Right : Complex_Vector) return Complex_Matrix;        36/2

function "*" (Left  : Complex_Vector;                                     37/2
              Right : Complex_Matrix) return Complex_Vector;
function "*" (Left  : Complex_Matrix;
              Right : Complex_Vector) return Complex_Vector;

-- Mixed Real_Matrix and Complex_Matrix arithmetic operations             38/2

function "+" (Left  : Real_Matrix;                                        39/2
              Right : Complex_Matrix) return Complex_Matrix;
function "+" (Left  : Complex_Matrix;
              Right : Real_Matrix)    return Complex_Matrix;
function "-" (Left  : Real_Matrix;
              Right : Complex_Matrix) return Complex_Matrix;
function "-" (Left  : Complex_Matrix;
              Right : Real_Matrix)    return Complex_Matrix;
function "*" (Left  : Real_Matrix;
              Right : Complex_Matrix) return Complex_Matrix;
function "*" (Left  : Complex_Matrix;
              Right : Real_Matrix)    return Complex_Matrix;
```

```
40/2        function "*" (Left  : Real_Vector;
                          Right : Complex_Vector) return Complex_Matrix;
            function "*" (Left  : Complex_Vector;
                          Right : Real_Vector)    return Complex_Matrix;
41/2        function "*" (Left  : Real_Vector;
                          Right : Complex_Matrix) return Complex_Vector;
            function "*" (Left  : Complex_Vector;
                          Right : Real_Matrix)    return Complex_Vector;
            function "*" (Left  : Real_Matrix;
                          Right : Complex_Vector) return Complex_Vector;
            function "*" (Left  : Complex_Matrix;
                          Right : Real_Vector)    return Complex_Vector;
42/2        -- Complex_Matrix scaling operations
43/2        function "*" (Left  : Complex;
                          Right : Complex_Matrix) return Complex_Matrix;
            function "*" (Left  : Complex_Matrix;
                          Right : Complex)        return Complex_Matrix;
            function "/" (Left  : Complex_Matrix;
                          Right : Complex)        return Complex_Matrix;
44/2        function "*" (Left  : Real'Base;
                          Right : Complex_Matrix) return Complex_Matrix;
            function "*" (Left  : Complex_Matrix;
                          Right : Real'Base)      return Complex_Matrix;
            function "/" (Left  : Complex_Matrix;
                          Right : Real'Base)      return Complex_Matrix;
45/2        -- Complex_Matrix inversion and related operations
46/2        function Solve (A : Complex_Matrix; X : Complex_Vector)
               return Complex_Vector;
            function Solve (A, X : Complex_Matrix) return Complex_Matrix;
            function Inverse (A : Complex_Matrix) return Complex_Matrix;
            function Determinant (A : Complex_Matrix) return Complex;
47/2        -- Eigenvalues and vectors of a Hermitian matrix
48/2        function Eigenvalues(A : Complex_Matrix) return Real_Vector;
49/2        procedure Eigensystem(A       : in  Complex_Matrix;
                                  Values  : out Real_Vector;
                                  Vectors : out Complex_Matrix);
50/2        -- Other Complex_Matrix operations
51/2        function Unit_Matrix (Order            : Positive;
                                  First_1, First_2 : Integer := 1)
                                    return Complex_Matrix;
52/2    end Ada.Numerics.Generic_Complex_Arrays;
```

53/2 {*AI95-00296-01*} The library package Numerics.Complex_Arrays is declared pure and defines the same types and subprograms as Numerics.Generic_Complex_Arrays, except that the predefined type Float is systematically substituted for Real'Base, and the Real_Vector and Real_Matrix types exported by Numerics.Real_Arrays are systematically substituted for Real_Vector and Real_Matrix, and the Complex type exported by Numerics.Complex_Types is systematically substituted for Complex, throughout. Nongeneric equivalents for each of the other predefined floating point types are defined similarly, with the names Numerics.Short_Complex_Arrays, Numerics.Long_Complex_Arrays, etc.

54/2 {*AI95-00296-01*} Two types are defined and exported by Numerics.Generic_Complex_Arrays. The composite type Complex_Vector is provided to represent a vector with components of type Complex; it is defined as an unconstrained one-dimensional array with an index of type Integer. The composite type Complex_Matrix is provided to represent a matrix with components of type Complex; it is defined as an unconstrained, two-dimensional array with indices of type Integer.

55/2 {*AI95-00296-01*} The effect of the various subprograms is as described below. In many cases they are described in terms of corresponding scalar operations in Numerics.Generic_Complex_Types. Any

exception raised by those operations is propagated by the array subprogram. Moreover, any constraints on the parameters and the accuracy of the result for each individual component are as defined for the scalar operation.

{*AI95-00296-01*} In the case of those operations which are defined to *involve an inner product*, Constraint_Error may be raised if an intermediate result has a component outside the range of Real'Base even though the final mathematical result would not. 56/2

Discussion: {*AI05-0047-1*} An inner product never involves implicit complex conjugation. If the product of a vector with the conjugate of another (or the same) vector is required, then this has to be stated explicitly by writing for example X * Conjugate(Y). This mimics the usual mathematical notation. 56.a/3

```
function Re (X : Complex_Vector) return Real_Vector;
function Im (X : Complex_Vector) return Real_Vector;
```
57/2

{*AI95-00296-01*} Each function returns a vector of the specified Cartesian components of X. The index range of the result is X'Range. 58/2

```
procedure Set_Re (X  : in out Complex_Vector; Re : in Real_Vector);
procedure Set_Im (X  : in out Complex_Vector; Im : in Real_Vector);
```
59/2

{*AI95-00296-01*} Each procedure replaces the specified (Cartesian) component of each of the components of X by the value of the matching component of Re or Im; the other (Cartesian) component of each of the components is unchanged. Constraint_Error is raised if X'Length is not equal to Re'Length or Im'Length. 60/2

```
function Compose_From_Cartesian (Re     : Real_Vector)
   return Complex_Vector;
function Compose_From_Cartesian (Re, Im : Real_Vector)
   return Complex_Vector;
```
61/2

{*AI95-00296-01*} Each function constructs a vector of Complex results (in Cartesian representation) formed from given vectors of Cartesian components; when only the real components are given, imaginary components of zero are assumed. The index range of the result is Re'Range. Constraint_Error is raised if Re'Length is not equal to Im'Length. 62/2

```
function Modulus  (X     : Complex_Vector) return Real_Vector;
function "abs"    (Right : Complex_Vector) return Real_Vector
                                              renames Modulus;
function Argument (X     : Complex_Vector) return Real_Vector;
function Argument (X     : Complex_Vector;
                   Cycle : Real'Base)      return Real_Vector;
```
63/2

{*AI95-00296-01*} Each function calculates and returns a vector of the specified polar components of X or Right using the corresponding function in numerics.generic_complex_types. The index range of the result is X'Range or Right'Range. 64/2

```
function Compose_From_Polar (Modulus, Argument : Real_Vector)
   return Complex_Vector;
function Compose_From_Polar (Modulus, Argument : Real_Vector;
                             Cycle             : Real'Base)
   return Complex_Vector;
```
65/2

{*AI95-00296-01*} Each function constructs a vector of Complex results (in Cartesian representation) formed from given vectors of polar components using the corresponding function in numerics.generic_complex_types on matching components of Modulus and Argument. The index range of the result is Modulus'Range. Constraint_Error is raised if Modulus'Length is not equal to Argument'Length. 66/2

67/2
```
function "+" (Right : Complex_Vector) return Complex_Vector;
function "-" (Right : Complex_Vector) return Complex_Vector;
```

68/2 {*AI95-00296-01*} Each operation returns the result of applying the corresponding operation in numerics.generic_complex_types to each component of Right. The index range of the result is Right'Range.

69/2
```
function Conjugate (X : Complex_Vector) return Complex_Vector;
```

70/2 {*AI95-00296-01*} This function returns the result of applying the appropriate function Conjugate in numerics.generic_complex_types to each component of X. The index range of the result is X'Range.

71/2
```
function "+" (Left, Right : Complex_Vector) return Complex_Vector;
function "-" (Left, Right : Complex_Vector) return Complex_Vector;
```

72/2 {*AI95-00296-01*} Each operation returns the result of applying the corresponding operation in numerics.generic_complex_types to each component of Left and the matching component of Right. The index range of the result is Left'Range. Constraint_Error is raised if Left'Length is not equal to Right'Length.

73/2
```
function "*" (Left, Right : Complex_Vector) return Complex;
```

74/2 {*AI95-00296-01*} This operation returns the inner product of Left and Right. Constraint_Error is raised if Left'Length is not equal to Right'Length. This operation involves an inner product.

75/3
```
{AI05-0047-1} function "abs" (Right : Complex_Vector) return Real'Base;
```

76/2 {*AI95-00418-01*} This operation returns the Hermitian L2-norm of Right (the square root of the inner product of the vector with its conjugate).

76.a/2 **Implementation Note:** While the definition is given in terms of an inner product, the norm doesn't "involve an inner product" in the technical sense. The reason is that it has accuracy requirements substantially different from those applicable to inner products; and that cancellations cannot occur, because all the terms are positive, so there is no possibility of intermediate overflow.

77/2
```
function "+" (Left  : Real_Vector;
              Right : Complex_Vector) return Complex_Vector;
function "+" (Left  : Complex_Vector;
              Right : Real_Vector)    return Complex_Vector;
function "-" (Left  : Real_Vector;
              Right : Complex_Vector) return Complex_Vector;
function "-" (Left  : Complex_Vector;
              Right : Real_Vector)    return Complex_Vector;
```

78/2 {*AI95-00296-01*} Each operation returns the result of applying the corresponding operation in numerics.generic_complex_types to each component of Left and the matching component of Right. The index range of the result is Left'Range. Constraint_Error is raised if Left'Length is not equal to Right'Length.

79/2
```
function "*" (Left : Real_Vector;    Right : Complex_Vector) return Complex;
function "*" (Left : Complex_Vector; Right : Real_Vector)    return Complex;
```

80/2 {*AI95-00296-01*} Each operation returns the inner product of Left and Right. Constraint_Error is raised if Left'Length is not equal to Right'Length. These operations involve an inner product.

81/2
```
function "*" (Left : Complex; Right : Complex_Vector) return Complex_Vector;
```

82/2 {*AI95-00296-01*} This operation returns the result of multiplying each component of Right by the complex number Left using the appropriate operation "*" in numerics.generic_complex_types. The index range of the result is Right'Range.

```
function "*" (Left : Complex_Vector; Right : Complex) return Complex_Vector;
function "/" (Left : Complex_Vector; Right : Complex) return Complex_Vector;
```
83/2

{*AI95-00296-01*} Each operation returns the result of applying the corresponding operation in numerics.generic_complex_types to each component of the vector Left and the complex number Right. The index range of the result is Left'Range.
84/2

```
function "*" (Left : Real'Base;
              Right : Complex_Vector) return Complex_Vector;
```
85/2

{*AI95-00296-01*} This operation returns the result of multiplying each component of Right by the real number Left using the appropriate operation "*" in numerics.generic_complex_types. The index range of the result is Right'Range.
86/2

```
function "*" (Left : Complex_Vector;
              Right : Real'Base) return Complex_Vector;
function "/" (Left : Complex_Vector;
              Right : Real'Base) return Complex_Vector;
```
87/2

{*AI95-00296-01*} Each operation returns the result of applying the corresponding operation in numerics.generic_complex_types to each component of the vector Left and the real number Right. The index range of the result is Left'Range.
88/2

```
function Unit_Vector (Index : Integer;
                      Order : Positive;
                      First : Integer := 1) return Complex_Vector;
```
89/2

{*AI95-00296-01*} This function returns a *unit vector* with Order components and a lower bound of First. All components are set to (0.0, 0.0) except for the Index component which is set to (1.0, 0.0). Constraint_Error is raised if Index < First, Index > First + Order – 1, or if First + Order – 1 > Integer'Last.
90/2

```
function Re (X : Complex_Matrix) return Real_Matrix;
function Im (X : Complex_Matrix) return Real_Matrix;
```
91/2

{*AI95-00296-01*} Each function returns a matrix of the specified Cartesian components of X. The index ranges of the result are those of X.
92/2

```
procedure Set_Re (X : in out Complex_Matrix; Re : in Real_Matrix);
procedure Set_Im (X : in out Complex_Matrix; Im : in Real_Matrix);
```
93/2

{*AI95-00296-01*} Each procedure replaces the specified (Cartesian) component of each of the components of X by the value of the matching component of Re or Im; the other (Cartesian) component of each of the components is unchanged. Constraint_Error is raised if X'Length(1) is not equal to Re'Length(1) or Im'Length(1) or if X'Length(2) is not equal to Re'Length(2) or Im'Length(2).
94/2

```
function Compose_From_Cartesian (Re     : Real_Matrix)
   return Complex_Matrix;
function Compose_From_Cartesian (Re, Im : Real_Matrix)
   return Complex_Matrix;
```
95/2

{*AI95-00296-01*} Each function constructs a matrix of Complex results (in Cartesian representation) formed from given matrices of Cartesian components; when only the real components are given, imaginary components of zero are assumed. The index ranges of the result are those of Re. Constraint_Error is raised if Re'Length(1) is not equal to Im'Length(1) or Re'Length(2) is not equal to Im'Length(2).
96/2

97/2
```
function Modulus  (X     : Complex_Matrix) return Real_Matrix;
function "abs"    (Right : Complex_Matrix) return Real_Matrix
                                              renames Modulus;
function Argument (X     : Complex_Matrix) return Real_Matrix;
function Argument (X     : Complex_Matrix;
                   Cycle : Real'Base)      return Real_Matrix;
```

98/2 {*AI95-00296-01*} Each function calculates and returns a matrix of the specified polar components of X or Right using the corresponding function in numerics.generic_complex_types. The index ranges of the result are those of X or Right.

99/2
```
function Compose_From_Polar (Modulus, Argument : Real_Matrix)
   return Complex_Matrix;
function Compose_From_Polar (Modulus, Argument : Real_Matrix;
                             Cycle             : Real'Base)
   return Complex_Matrix;
```

100/2 {*AI95-00296-01*} Each function constructs a matrix of Complex results (in Cartesian representation) formed from given matrices of polar components using the corresponding function in numerics.generic_complex_types on matching components of Modulus and Argument. The index ranges of the result are those of Modulus. Constraint_Error is raised if Modulus'Length(1) is not equal to Argument'Length(1) or Modulus'Length(2) is not equal to Argument'Length(2).

101/2
```
function "+" (Right : Complex_Matrix) return Complex_Matrix;
function "-" (Right : Complex_Matrix) return Complex_Matrix;
```

102/2 {*AI95-00296-01*} Each operation returns the result of applying the corresponding operation in numerics.generic_complex_types to each component of Right. The index ranges of the result are those of Right.

103/2
```
function Conjugate (X : Complex_Matrix) return Complex_Matrix;
```

104/2 {*AI95-00296-01*} This function returns the result of applying the appropriate function Conjugate in numerics.generic_complex_types to each component of X. The index ranges of the result are those of X.

105/2
```
function Transpose (X : Complex_Matrix) return Complex_Matrix;
```

106/2 {*AI95-00296-01*} This function returns the transpose of a matrix X. The first and second index ranges of the result are X'Range(2) and X'Range(1) respectively.

107/2
```
function "+" (Left, Right : Complex_Matrix) return Complex_Matrix;
function "-" (Left, Right : Complex_Matrix) return Complex_Matrix;
```

108/2 {*AI95-00296-01*} Each operation returns the result of applying the corresponding operation in numerics.generic_complex_types to each component of Left and the matching component of Right. The index ranges of the result are those of Left. Constraint_Error is raised if Left'Length(1) is not equal to Right'Length(1) or Left'Length(2) is not equal to Right'Length(2).

109/2
```
function "*" (Left, Right : Complex_Matrix) return Complex_Matrix;
```

110/2 {*AI95-00296-01*} This operation provides the standard mathematical operation for matrix multiplication. The first and second index ranges of the result are Left'Range(1) and Right'Range(2) respectively. Constraint_Error is raised if Left'Length(2) is not equal to Right'Length(1). This operation involves inner products.

111/2
```
function "*" (Left, Right : Complex_Vector) return Complex_Matrix;
```

112/2 {*AI95-00296-01*} This operation returns the outer product of a (column) vector Left by a (row) vector Right using the appropriate operation "*" in numerics.generic_complex_types for computing the individual components. The first and second index ranges of the result are Left'Range and Right'Range respectively.

```
function "*" (Left  : Complex_Vector;
              Right : Complex_Matrix) return Complex_Vector;
```
113/2

{*AI95-00296-01*} This operation provides the standard mathematical operation for multiplication of a (row) vector Left by a matrix Right. The index range of the (row) vector result is Right'Range(2). Constraint_Error is raised if Left'Length is not equal to Right'Length(1). This operation involves inner products.
114/2

```
function "*" (Left  : Complex_Matrix;
              Right : Complex_Vector) return Complex_Vector;
```
115/2

{*AI95-00296-01*} This operation provides the standard mathematical operation for multiplication of a matrix Left by a (column) vector Right. The index range of the (column) vector result is Left'Range(1). Constraint_Error is raised if Left'Length(2) is not equal to Right'Length. This operation involves inner products.
116/2

```
function "+" (Left  : Real_Matrix;
              Right : Complex_Matrix) return Complex_Matrix;
function "+" (Left  : Complex_Matrix;
              Right : Real_Matrix)    return Complex_Matrix;
function "-" (Left  : Real_Matrix;
              Right : Complex_Matrix) return Complex_Matrix;
function "-" (Left  : Complex_Matrix;
              Right : Real_Matrix)    return Complex_Matrix;
```
117/2

{*AI95-00296-01*} Each operation returns the result of applying the corresponding operation in numerics.generic_complex_types to each component of Left and the matching component of Right. The index ranges of the result are those of Left. Constraint_Error is raised if Left'Length(1) is not equal to Right'Length(1) or Left'Length(2) is not equal to Right'Length(2).
118/2

```
function "*" (Left  : Real_Matrix;
              Right : Complex_Matrix) return Complex_Matrix;
function "*" (Left  : Complex_Matrix;
              Right : Real_Matrix)    return Complex_Matrix;
```
119/2

{*AI95-00296-01*} Each operation provides the standard mathematical operation for matrix multiplication. The first and second index ranges of the result are Left'Range(1) and Right'Range(2) respectively. Constraint_Error is raised if Left'Length(2) is not equal to Right'Length(1). These operations involve inner products.
120/2

```
function "*" (Left  : Real_Vector;
              Right : Complex_Vector) return Complex_Matrix;
function "*" (Left  : Complex_Vector;
              Right : Real_Vector)    return Complex_Matrix;
```
121/2

{*AI95-00296-01*} Each operation returns the outer product of a (column) vector Left by a (row) vector Right using the appropriate operation "*" in numerics.generic_complex_types for computing the individual components. The first and second index ranges of the result are Left'Range and Right'Range respectively.
122/2

```
function "*" (Left  : Real_Vector;
              Right : Complex_Matrix) return Complex_Vector;
function "*" (Left  : Complex_Vector;
              Right : Real_Matrix)    return Complex_Vector;
```
123/2

{*AI95-00296-01*} Each operation provides the standard mathematical operation for multiplication of a (row) vector Left by a matrix Right. The index range of the (row) vector result is Right'Range(2). Constraint_Error is raised if Left'Length is not equal to Right'Length(1). These operations involve inner products.
124/2

125/2
```
function "*" (Left  : Real_Matrix;
              Right : Complex_Vector) return Complex_Vector;
function "*" (Left  : Complex_Matrix;
              Right : Real_Vector)    return Complex_Vector;
```

126/2 {*AI95-00296-01*} Each operation provides the standard mathematical operation for multiplication of a matrix Left by a (column) vector Right. The index range of the (column) vector result is Left'Range(1). Constraint_Error is raised if Left'Length(2) is not equal to Right'Length. These operations involve inner products.

127/2
```
function "*" (Left : Complex; Right : Complex_Matrix) return Complex_Matrix;
```

128/2 {*AI95-00296-01*} This operation returns the result of multiplying each component of Right by the complex number Left using the appropriate operation "*" in numerics.generic_complex_types. The index ranges of the result are those of Right.

129/2
```
function "*" (Left : Complex_Matrix; Right : Complex) return Complex_Matrix;
function "/" (Left : Complex_Matrix; Right : Complex) return Complex_Matrix;
```

130/2 {*AI95-00296-01*} Each operation returns the result of applying the corresponding operation in numerics.generic_complex_types to each component of the matrix Left and the complex number Right. The index ranges of the result are those of Left.

131/2
```
function "*" (Left : Real'Base;
              Right : Complex_Matrix) return Complex_Matrix;
```

132/2 {*AI95-00296-01*} This operation returns the result of multiplying each component of Right by the real number Left using the appropriate operation "*" in numerics.generic_complex_types. The index ranges of the result are those of Right.

133/2
```
function "*" (Left : Complex_Matrix;
              Right : Real'Base) return Complex_Matrix;
function "/" (Left : Complex_Matrix;
              Right : Real'Base) return Complex_Matrix;
```

134/2 {*AI95-00296-01*} Each operation returns the result of applying the corresponding operation in numerics.generic_complex_types to each component of the matrix Left and the real number Right. The index ranges of the result are those of Left.

135/2
```
function Solve (A : Complex_Matrix; X : Complex_Vector) return Complex_Vector;
```

136/2 {*AI95-00296-01*} This function returns a vector Y such that X is (nearly) equal to A * Y. This is the standard mathematical operation for solving a single set of linear equations. The index range of the result is A'Range(2). Constraint_Error is raised if A'Length(1), A'Length(2), and X'Length are not equal. Constraint_Error is raised if the matrix A is ill-conditioned.

136.a/2 **Discussion:** The text says that Y is such that “X is (nearly) equal to A * Y” rather than “X is equal to A * Y” because rounding errors may mean that there is no value of Y such that X is exactly equal to A * Y. On the other hand it does not mean that any old rough value will do. The algorithm given under Implementation Advice should be followed.

136.b/2 The requirement to raise Constraint_Error if the matrix is ill-conditioned is really a reflection of what will happen if the matrix is ill-conditioned. See Implementation Advice. We do not make any attempt to define ill-conditioned formally.

136.c/2 These remarks apply to all versions of Solve and Inverse.

137/2
```
function Solve (A, X : Complex_Matrix) return Complex_Matrix;
```

138/2 {*AI95-00296-01*} This function returns a matrix Y such that X is (nearly) equal to A * Y. This is the standard mathematical operation for solving several sets of linear equations. The index ranges of the result are A'Range(2) and X'Range(2). Constraint_Error is raised if A'Length(1), A'Length(2), and X'Length(1) are not equal. Constraint_Error is raised if the matrix A is ill-conditioned.

```
function Inverse (A : Complex_Matrix) return Complex_Matrix;
```
139/2

{*AI95-00296-01*} This function returns a matrix B such that A * B is (nearly) equal to the unit matrix. The index ranges of the result are A'Range(2) and A'Range(1). Constraint_Error is raised if A'Length(1) is not equal to A'Length(2). Constraint_Error is raised if the matrix A is ill-conditioned.
140/2

```
function Determinant (A : Complex_Matrix) return Complex;
```
141/2

{*AI95-00296-01*} This function returns the determinant of the matrix A. Constraint_Error is raised if A'Length(1) is not equal to A'Length(2).
142/2

```
function Eigenvalues(A : Complex_Matrix) return Real_Vector;
```
143/2

{*AI95-00296-01*} This function returns the eigenvalues of the Hermitian matrix A as a vector sorted into order with the largest first. Constraint_Error is raised if A'Length(1) is not equal to A'Length(2). The index range of the result is A'Range(1). Argument_Error is raised if the matrix A is not Hermitian.
144/2

Discussion: A Hermitian matrix is one whose transpose is equal to its complex conjugate. The eigenvalues of a Hermitian matrix are always real. We only support this case because algorithms for solving the general case are inherently unstable.
144.a/2

```
procedure Eigensystem(A       : in  Complex_Matrix;
                      Values  :  out Real_Vector;
                      Vectors :  out Complex_Matrix);
```
145/2

{*AI95-00296-01*} {*AI05-0047-1*} This procedure computes both the eigenvalues and eigenvectors of the Hermitian matrix A. The out parameter Values is the same as that obtained by calling the function Eigenvalues. The out parameter Vectors is a matrix whose columns are the eigenvectors of the matrix A. The order of the columns corresponds to the order of the eigenvalues. The eigenvectors are mutually orthonormal, including when there are repeated eigenvalues. Constraint_Error is raised if A'Length(1) is not equal to A'Length(2), or if Values'Range is not equal to A'Range(1), or if the index ranges of the parameter Vectors are not equal to those of A. Argument_Error is raised if the matrix A is not Hermitian. Constraint_Error is also raised in implementation-defined circumstances if the algorithm used does not converge quickly enough.
146/3

Ramification: {*AI05-0047-1*} There is no requirement on the absolute direction of the returned eigenvectors. Thus they might be multiplied by any complex number whose modulus is 1. It is only the ratios of the components that matter. This is standard practice.
146.a/3

```
function Unit_Matrix (Order            : Positive;
                      First_1, First_2 : Integer := 1)
                                         return Complex_Matrix;
```
147/2

{*AI95-00296-01*} This function returns a square *unit matrix* with Order**2 components and lower bounds of First_1 and First_2 (for the first and second index ranges respectively). All components are set to (0.0, 0.0) except for the main diagonal, whose components are set to (1.0, 0.0). Constraint_Error is raised if First_1 + Order – 1 > Integer'Last or First_2 + Order – 1 > Integer'Last.
148/2

Implementation Requirements

{*AI95-00296-01*} Accuracy requirements for the subprograms Solve, Inverse, Determinant, Eigenvalues and Eigensystem are implementation defined.
149/2

Implementation defined: The accuracy requirements for the subprograms Solve, Inverse, Determinant, Eigenvalues and Eigensystem for type Complex_Matrix.
149.a/2

{*AI95-00296-01*} For operations not involving an inner product, the accuracy requirements are those of the corresponding operations of the type Real'Base and Complex in both the strict mode and the relaxed mode (see G.2).
150/2

151/2 {*AI95-00296-01*} For operations involving an inner product, no requirements are specified in the relaxed mode. In the strict mode the modulus of the absolute error of the inner product *X***Y* shall not exceed *g****abs**(*X*)***abs**(*Y*) where *g* is defined as

152/2 *g* = *X*'Length * Real'Machine_Radix**(1 – Real'Model_Mantissa)
for mixed complex and real operands

153/2 *g* = sqrt(2.0) * *X*'Length * Real'Machine_Radix**(1 – Real'Model_Mantissa)
for two complex operands

154/2 {*AI95-00418-01*} For the L2-norm, no accuracy requirements are specified in the relaxed mode. In the strict mode the relative error on the norm shall not exceed *g* / 2.0 + 3.0 * Real'Model_Epsilon where *g* has the definition appropriate for two complex operands.

Documentation Requirements

155/2 {*AI95-00296-01*} Implementations shall document any techniques used to reduce cancellation errors such as extended precision arithmetic.

155.a/2 **Documentation Requirement:** Any techniques used to reduce cancellation errors in Numerics.Generic_Complex_Arrays shall be documented.

155.b/2 **Implementation Note:** The above accuracy requirement is met by the canonical implementation of the inner product by multiplication and addition using the corresponding operations of type Complex and performing the cumulative addition using ascending indices. Note however, that some hardware provides special operations for the computation of the inner product and although these may be fast they may not meet the accuracy requirement specified. See Accuracy and Stability of Numerical Algorithms by N J Higham (ISBN 0-89871-355-2), Sections 3.1 and 3.6.

Implementation Permissions

156/5 {*AI95-00296-01*} {*AI12-0444-1*} The nongeneric equivalent packages can be actual instantiations of the generic package for the appropriate predefined type, though that is not required.

157/5 {*AI95-00296-01*} {*AI12-0444-1*} Although many operations are defined in terms of operations from Numerics.Generic_Complex_Types, they can be implemented by other operations that have the same effect.

Implementation Advice

158/3 {*AI95-00296-01*} {*AI05-0264-1*} Implementations should implement the Solve and Inverse functions using established techniques. Implementations are recommended to refine the result by performing an iteration on the residuals; if this is done, then it should be documented.

158.a/2 **Implementation Advice:** Solve and Inverse for Numerics.Generic_Complex_Arrays should be implemented using established techniques and the result should be refined by an iteration on the residuals.

159/2 {*AI95-00296-01*} It is not the intention that any special provision should be made to determine whether a matrix is ill-conditioned or not. The naturally occurring overflow (including division by zero) which will result from executing these functions with an ill-conditioned matrix and thus raise Constraint_Error is sufficient.

159.a/2 **Discussion:** There isn't any advice for the implementation to document with this paragraph.

160/2 {*AI95-00296-01*} The test that a matrix is Hermitian should use the equality operator to compare the real components and negation followed by equality to compare the imaginary components (see G.2.1).

160.a/2 **Implementation Advice:** The equality and negation operators should be used to test that a matrix is Hermitian.

160.1/3 An implementation should minimize the circumstances under which the algorithm used for Eigenvalues and Eigensystem fails to converge.

Implementation Advice: An implementation should minimize the circumstances under which the algorithm used for Numerics.Generic_Complex_Arrays.Eigenvalues and Numerics.Generic_Complex_Arrays.Eigensystem fails to converge. 160.a.1/3

Implementation Note: J. H. Wilkinson is the acknowledged expert in this area. See for example Wilkinson, J. H., and Reinsch, C. , Linear Algebra , vol II of Handbook for Automatic Computation, Springer-Verlag, or Wilkinson, J. H., The Algebraic Eigenvalue Problem, Oxford University Press. 160.b/3

{*AI95-00296-01*} Implementations should not perform operations on mixed complex and real operands by first converting the real operand to complex. See G.1.1. 161/2

Implementation Advice: Mixed real and complex operations should not be performed by converting the real operand to complex. 161.a/2

Extensions to Ada 95

{*AI95-00296-01*} The package Numerics.Generic_Complex_Arrays and its nongeneric equivalents are new. 161.b/2

Wording Changes from Ada 2005

{*AI05-0047-1*} **Correction:** Corrected various accuracy and definition issues. 161.c/3

Annex H
(normative)
High Integrity Systems

{*AI95-00347-01*} This Annex addresses requirements for high integrity systems (including safety-critical systems and security-critical systems). It provides facilities and specifies documentation requirements that relate to several needs: 1/2

- Understanding program execution; 2
- Reviewing object code; 3
- {*AI12-0439-1*} Restricting language constructs whose usage can complicate the demonstration of program correctness 4/5

Execution understandability is supported by pragma Normalize_Scalars, and also by requirements for the implementation to document the effect of a program in the presence of a bounded error or where the language rules leave the effect unspecified. 4.1

The pragmas Reviewable and Restrictions relate to the other requirements addressed by this Annex. 5

NOTE {*AI12-0440-1*} The Valid attribute (see 13.9.2) is also useful in addressing these needs, to avoid problems that can otherwise arise from scalars that have values outside their declared range constraints. 6/5

Discussion: The Annex tries to provide high assurance rather than language features. However, it is not possible, in general, to test for high assurance. For any specific language feature, it is possible to demonstrate its presence by a functional test, as in the ACVC. One can also check for the presence of some documentation requirements, but it is not easy to determine objectively that the documentation is “adequate”. 6.a

Extensions to Ada 83

This Annex is new to Ada 95. 6.b

Wording Changes from Ada 95

{*AI95-00347-01*} The title of this annex was changed to better reflect its purpose and scope. High integrity systems has become the standard way of identifying systems that have high reliability requirements; it subsumes terms such as safety and security. Moreover, the annex does not include any security specific features and as such the previous title is somewhat misleading. 6.c/2

H.1 Pragma Normalize_Scalars

This pragma ensures that an otherwise uninitialized scalar object is set to a predictable value, but out of range if possible. 1

Discussion: The goal of the pragma is to reduce the impact of a bounded error that results from a reference to an uninitialized scalar object, by having such a reference violate a range check and thus raise Constraint_Error. 1.a

Syntax

The form of a pragma Normalize_Scalars is as follows: 2

pragma Normalize_Scalars; 3

Post-Compilation Rules

Pragma Normalize_Scalars is a configuration pragma. It applies to all compilation_units included in a partition. 4

Documentation Requirements

5/2 {*AI95-00434-01*} If a pragma Normalize_Scalars applies, the implementation shall document the implicit initial values for scalar subtypes, and shall identify each case in which such a value is used and is not an invalid representation.

5.a/2 **Documentation Requirement:** If a pragma Normalize_Scalars applies, the implicit initial values of scalar subtypes shall be documented. Such a value should be an invalid representation when possible; any cases when is it not shall be documented.

5.b **To be honest:** It's slightly inaccurate to say that the value is a representation, but the point should be clear anyway.

5.c **Discussion:** By providing a type with a size specification so that spare bits are present, it is possible to force an implementation of Normalize_Scalars to use an out of range value. This can be tested for by ensuring that Constraint_Error is raised. Similarly, for an unconstrained integer type, in which no spare bit is surely present, one can check that the initialization takes place to the value specified in the documentation of the implementation. For a floating point type, spare bits might not be available, but a range constraint can provide the ability to use an out of range value.

5.d If it is difficult to document the general rule for the implicit initial value, the implementation might choose instead to record the value on the object code listing or similar output produced during compilation.

Implementation Advice

6/2 {*AI95-00434-01*} Whenever possible, the implicit initial values for a scalar subtype should be an invalid representation (see 13.9.1).

6.a **Discussion:** When an out of range value is used for the initialization, it is likely that constraint checks will detect it. In addition, it can be detected by the Valid attribute.

6.b/2 This rule is included in the documentation requirements, and thus does not need a separate summary item.

7 NOTE 1 The initialization requirement applies to uninitialized scalar objects that are subcomponents of composite objects, to allocated objects, and to stand-alone objects. It also applies to scalar **out** parameters. Scalar subcomponents of composite **out** parameters are initialized to the corresponding part of the actual, by virtue of 6.4.1.

8 NOTE 2 The initialization requirement does not apply to a scalar for which pragma Import has been specified, since initialization of an imported object is performed solely by the foreign language environment (see B.1).

9/5 NOTE 3 {*AI12-0440-1*} The use of pragma Normalize_Scalars in conjunction with Pragma Restrictions(No_Exceptions) can result in erroneous execution (see H.4).

9.a **Discussion:** Since the effect of an access to an out of range value will often be to raise Constraint_Error, it is clear that suppressing the exception mechanism could result in erroneous execution. In particular, the assignment to an array, with the array index out of range, will result in a write to an arbitrary store location, having unpredictable effects.

H.2 Documentation of Implementation Decisions

Documentation Requirements

1/5 {*AI12-0439-1*} The implementation shall document the range of effects for each situation that the language rules identify as either a bounded error or as having an unspecified effect. If the implementation can constrain the effects of erroneous execution for a given construct, then it shall document such constraints. [The documentation may be provided either independently of any compilation unit or partition, or as part of an annotated listing for a given unit or partition. See also 1.1.3, and 1.1.2.]

1.a/2 *This paragraph was deleted.*

1.b/2 **Documentation Requirement:** The range of effects for each bounded error and each unspecified effect. If the effects of a given erroneous construct are constrained, the constraints shall be documented.

2 NOTE Among the situations to be documented are the conventions chosen for parameter passing, the methods used for the management of run-time storage, and the method used to evaluate numeric expressions if this involves extended range or extra precision.

2.a **Discussion:** Look up “unspecified” and “erroneous execution” in the index for a list of the cases.

The management of run-time storage is particularly important. For safety applications, it is often necessary to show that a program cannot raise Storage_Error, and for security applications that information cannot leak via the run-time system. Users are likely to prefer a simple storage model that can be easily validated. 2.b

The documentation could helpfully take into account that users may well adopt a subset to avoid some forms of erroneous execution, for instance, not using the abort statement, so that the effects of a partly completed assignment_statement do not have to be considered in the validation of a program (see 9.8). For this reason documentation linked to an actual compilation may be most useful. Similarly, an implementation may be able to take into account use of the Restrictions pragma. 2.c

H.3 Reviewable Object Code

Object code review and validation are supported by pragmas Reviewable and Inspection_Point. 1

H.3.1 Pragma Reviewable

This pragma directs the implementation to provide information to facilitate analysis and review of a program's object code, in particular to allow determination of execution time and storage usage and to identify the correspondence between the source and object programs. 1

Discussion: Since the purpose of this pragma is to provide information to the user, it is hard to objectively test for conformity. In practice, users want the information in an easily understood and convenient form, but neither of these properties can be easily measured. 1.a

Syntax

The form of a pragma Reviewable is as follows: 2

pragma Reviewable; 3

Post-Compilation Rules

Pragma Reviewable is a configuration pragma. It applies to all compilation_units included in a partition. 4

Implementation Requirements

The implementation shall provide the following information for any compilation unit to which such a pragma applies: 5

Discussion: The list of requirements can be checked for, even if issues like intelligibility are not addressed. 5.a

- Where compiler-generated runtime checks remain; 6

  Discussion: A constraint check which is implemented via a check on the upper and lower bound should clearly be indicated. If a check is implicit in the form of machine instructions used (such an overflow checking), this should also be covered by the documentation. It is particularly important to cover those checks which are not obvious from the source code, such as that for stack overflow. 6.a

- An identification of any construct with a language-defined check that is recognized prior to run time as certain to fail if executed (even if the generation of runtime checks has been suppressed); 7

  Discussion: In this case, if the compiler determines that a check must fail, the user should be informed of this. However, since it is not in general possible to know what the compiler will detect, it is not easy to test for this. In practice, it is thought that compilers claiming conformity to this Annex will perform significant optimizations and therefore *will* detect such situations. Of course, such events could well indicate a programmer error. 7.a

- {*AI95-00209-01*} For each read of a scalar object, an identification of the read as either “known to be initialized”, or “possibly uninitialized”, independent of whether pragma Normalize_Scalars applies; 8/2

  Discussion: This issue again raises the question as to what the compiler has determined. A lazy implementation could clearly mark all scalars as “possibly uninitialized”, but this would be very unhelpful to the user. It should be possible to analyze a range of scalar uses and note the percentage in each class. Note that an access marked “known to be initialized” does not imply that the value is in range, since the initialization could be from an (erroneous) call of unchecked conversion, or by means external to the Ada program. 8.a

9 • Where run-time support routines are implicitly invoked;

9.a **Discussion:** Validators will need to know the calls invoked in order to check for the correct functionality. For instance, for some safety applications, it may be necessary to ensure that certain sections of code can execute in a particular time.

10 • An object code listing, including:

11 • Machine instructions, with relative offsets;

11.a **Discussion:** The machine instructions should be in a format that is easily understood, such as the symbolic format of the assembler. The relative offsets are needed in numeric format, to check any alignment restrictions that the architecture might impose.

12 • Where each data object is stored during its lifetime;

12.a **Discussion:** This requirement implies that if the optimizer assigns a variable to a register, this needs to be evident.

13 • Correspondence with the source program, including an identification of the code produced per declaration and per statement.

13.a **Discussion:** This correspondence will be quite complex when extensive optimization is performed. In particular, address calculation to access some data structures could be moved from the actual access. However, when all the machine code arising from a statement or declaration is in one basic block, this must be indicated by the implementation.

14 • An identification of each construct for which the implementation detects the possibility of erroneous execution;

14.a **Discussion:** This requirement is quite vague. In general, it is hard for compilers to detect erroneous execution and therefore the requirement will be rarely invoked. However, if the pragma Suppress is used and the compiler can show that a predefined exception will be raised, then such an identification would be useful.

15 • For each subprogram, block, task, or other construct implemented by reserving and subsequently freeing an area on a run-time stack, an identification of the length of the fixed-size portion of the area and an indication of whether the non-fixed size portion is reserved on the stack or in a dynamically-managed storage region.

15.a **Discussion:** This requirement is vital for those requiring to show that the storage available to a program is sufficient. This is crucial in those cases in which the internal checks for stack overflow are suppressed (perhaps by **pragma** Restrictions(No_Exceptions)).

16 The implementation shall provide the following information for any partition to which the pragma applies:

17 • An object code listing of the entire partition, including initialization and finalization code as well as run-time system components, and with an identification of those instructions and data that will be relocated at load time;

17.a **Discussion:** The object code listing should enable a validator to estimate upper bounds for the time taken by critical parts of a program. Similarly, by an analysis of the entire partition, it should be possible to ensure that the storage requirements are suitably bounded, assuming that the partition was written in an appropriate manner.

18 • A description of the run-time model relevant to the partition.

18.a **Discussion:** For example, a description of the storage model is vital, since the Ada language does not explicitly define such a model.

18.1 The implementation shall provide control- and data-flow information, both within each compilation unit and across the compilation units of the partition.

18.b **Discussion:** This requirement is quite vague, since it is unclear what control and data flow information the compiler has produced. It is really a plea not to throw away information that could be useful to the validator. Note that the data flow information is relevant to the detection of "possibly uninitialized" objects referred to above.

Implementation Advice

19 The implementation should provide the above information in both a human-readable and machine-readable form, and should document the latter so as to ease further processing by automated tools.

Implementation Advice: The information produced by pragma Reviewable should be provided in both a human-readable and machine-readable form, and the latter form should be documented. 19.a/2

Object code listings should be provided both in a symbolic format and also in an appropriate numeric format (such as hexadecimal or octal). 20

Implementation Advice: Object code listings should be provided both in a symbolic format and in a numeric format. 20.a/2

Reason: This is to enable other tools to perform any analysis that the user needed to aid validation. The format should be in some agreed form. 20.b

NOTE The order of elaboration of library units will be documented even in the absence of pragma Reviewable (see 10.2). 21

Discussion: There might be some interactions between pragma Reviewable and compiler optimizations. For example, an implementation may disable some optimizations when pragma Reviewable is in force if it would be overly complicated to provide the detailed information to allow review of the optimized object code. See also pragma Optimize (2.8). 21.a

Wording Changes from Ada 95

{*AI95-00209-01*} The wording was clarified that pragma Reviewable applies to each read of an object, as it makes no sense to talk about the state of an object that will immediately be overwritten. 21.b/2

H.3.2 Pragma Inspection_Point

An occurrence of a pragma Inspection_Point identifies a set of objects each of whose values is to be available at the point(s) during program execution corresponding to the position of the pragma in the compilation unit. The purpose of such a pragma is to facilitate code validation. 1

Discussion: Inspection points are a high level equivalent of break points used by debuggers. 1.a

Syntax

The form of a pragma Inspection_Point is as follows: 2

pragma Inspection_Point[(*object*_name {, *object*_name})]; 3

Legality Rules

A pragma Inspection_Point is allowed wherever a declarative_item or statement is allowed. Each *object*_name shall statically denote the declaration of an object. 4

Discussion: The static denotation is required, since no dynamic evaluation of a name is involved in this pragma. 4.a

Static Semantics

{*8652/0093*} {*AI95-00207-01*} {*AI95-00434-01*} An *inspection point* is a point in the object code corresponding to the occurrence of a pragma Inspection_Point in the compilation unit. An object is *inspectable* at an inspection point if the corresponding pragma Inspection_Point either has an argument denoting that object, or has no arguments and the declaration of the object is visible at the inspection point. 5/2

Ramification: If a pragma Inspection_Point is in an in-lined subprogram, there might be numerous inspection points in the object code corresponding to the single occurrence of the pragma in the source; similar considerations apply if such a pragma is in a generic, or in a loop that has been "unrolled" by an optimizer. 5.a

{*8652/0093*} {*AI95-00207-01*} The short form of the pragma is a convenient shorthand for listing all objects which could be explicitly made inspectable by the long form of the pragma; thus only visible objects are made inspectable by it. Objects that are not visible at the point of the pragma are not made inspectable by the short form pragma. This is necessary so that implementations need not keep information about (or prevent optimizations on) a unit simply because some other unit *might* contain a short form Inspection_Point pragma. 5.a.1/1

Discussion: {*8652/0093*} {*AI95-00207-01*} If the short form of the pragma is used, then all visible objects are inspectable. This implies that global objects from other compilation units are inspectable. A good interactive debugging system could provide information similar to a post-mortem dump at such inspection points. The annex does 5.b/1

not require that any inspection facility is provided, merely that the information is available to understand the state of the machine at those points.

Dynamic Semantics

6 Execution of a pragma Inspection_Point has no effect.

6.a/2 **Discussion:** {*AI95-00114-01*} Although an inspection point has no (semantic) effect, the removal or adding of a new point could change the machine code generated by the compiler.

Implementation Requirements

7 Reaching an inspection point is an external interaction with respect to the values of the inspectable objects at that point (see 1.1.3).

7.a **Ramification:** The compiler is inhibited from moving an assignment to an inspectable variable past an inspection point for that variable. On the other hand, the evaluation of an expression that might raise an exception may be moved past an inspection point (see 11.6).

Documentation Requirements

8 For each inspection point, the implementation shall identify a mapping between each inspectable object and the machine resources (such as memory locations or registers) from which the object's value can be obtained.

8.a/2 *This paragraph was deleted.*

8.b/2 **Documentation Requirement:** For each inspection point, a mapping between each inspectable object and the machine resources where the object's value can be obtained shall be provided.

9/5 NOTE 1 {*AI95-00209-01*} {*AI12-0447-1*} Because reaching an inspection point is considered an external interaction relative to the values of the inspectable variables, the implementation cannot perform “dead store elimination” on the last assignment to such a variable prior to an inspection point. Thus an inspection point has the effect of an implicit read of each of its inspectable objects.

10 NOTE 2 Inspection points are useful in maintaining a correspondence between the state of the program in source code terms, and the machine state during the program's execution. Assertions about the values of program objects can be tested in machine terms at inspection points. Object code between inspection points can be processed by automated tools to verify programs mechanically.

10.a **Discussion:** Although it is not a requirement of the annex, it would be useful if the state of the stack and heap could be interrogated. This would allow users to check that a program did not have a `storage leak'.

11/5 NOTE 3 {*AI12-0447-1*} The identification of the mapping from source program objects to machine resources can be in the form of an annotated object listing, in human-readable or tool-processable form.

11.a **Discussion:** In principle, it is easy to check an implementation for this pragma, since one merely needs to check the content of objects against those values known from the source listing. In practice, one needs a tool similar to an interactive debugger to perform the check.

Wording Changes from Ada 95

11.b/2 {*8652/0093*} {*AI95-00207-01*} **Corrigendum:** Corrected the definition of the Inspection_Point pragma to apply to only variables visible at the point of the pragma. Otherwise, the compiler would have to assume that some other code somewhere could have a pragma Inspection_Point, preventing many optimizations (such as unused object elimination).

H.4 High Integrity Restrictions

1/3 {*AI05-0299-1*} This subclause defines restrictions that can be used with pragma Restrictions (see 13.12); these facilitate the demonstration of program correctness by allowing tailored versions of the run-time system.

1.a/3 **Discussion:** {*AI05-0005-1*} Note that the restrictions are absolute. If a partition has 100 library units and just one needs Unchecked_Conversion, then the pragma cannot be used to ensure the other 99 units do not use Unchecked_Conversion. Note also that these are restrictions on all Ada code within a partition, and therefore it might not be evident from the specification of a package whether a restriction can be imposed.

Static Semantics

This paragraph was deleted.{*AI95-00347-01*} {*AI95-00394-01*} 2/2

{*AI95-00394-01*} The following *restriction_*identifiers are language defined: 3/2

Tasking-related restriction: 4

No_Protected_Types 5
There are no declarations of protected types or protected objects.

Memory-management related restrictions: 6

No_Allocators 7
There are no occurrences of an allocator.

{*8652/0042*} {*AI95-00130*} No_Local_Allocators 8/1
Allocators are prohibited in subprograms, generic subprograms, tasks, and entry bodies.

Ramification: Thus allocators are permitted only in expressions whose evaluation can only be performed before the main subprogram is invoked. 8.a

This paragraph was deleted.{*8652/0042*} {*AI95-00130*} 8.b/1

{*AI05-0152-1*} {*AI05-0262-1*} No_Anonymous_Allocators 8.1/3
There are no allocators of anonymous access types.

{*AI05-0190-1*} No_Coextensions 8.2/3
There are no coextensions. See 3.10.2.

{*AI05-0190-1*} No_Access_Parameter_Allocators 8.3/3
Allocators are not permitted as the actual parameter to an access parameter. See 6.1.

This paragraph was deleted.{*AI95-00394-01*} 9/2

Immediate_Reclamation 10
Except for storage occupied by objects created by allocators and not deallocated via unchecked deallocation, any storage reserved at run time for an object is immediately reclaimed when the object no longer exists.

Discussion: Immediate reclamation would apply to storage created by the compiler, such as for a return value from a function whose size is not known at the call site. 10.a

Exception-related restriction: 11

{*AI12-0344-1*} No_Exceptions 12/5
Raise_statements and exception_handlers are not allowed. No language-defined runtime checks are generated; however, a runtime check performed automatically by the hardware is permitted. The callable entity associated with a procedural_iterator (see 5.5.3) is considered to not allow exit, independent of the value of its Allows_Exit aspect.

Discussion: This restriction mirrors a method of working that is quite common in the safety area. The programmer is required to show that exceptions cannot be raised. Then a simplified run-time system is used without exception handling. However, some hardware checks may still be enforced. If the software check would have failed, or if the hardware check actually fails, then the execution of the program is unpredictable. There are obvious dangers in this approach, but it is similar to programming at the assembler level. 12.a

Other restrictions: 13

No_Floating_Point 14
Uses of predefined floating point types and operations, and declarations of new floating point types, are not allowed.

Discussion: {*AI95-00114-01*} The intention is to avoid the use of floating point hardware at run time, but this is expressed in language terms. It is conceivable that floating point is used implicitly in some contexts, say fixed point type conversions of high accuracy. However, the Implementation Requirements below make it clear that the restriction 14.a/2

would apply to the “run-time system” and hence not be allowed. This restriction could be used to inform a compiler that a variant of the architecture is being used which does not have floating point instructions.

15 No_Fixed_Point
Uses of predefined fixed point types and operations, and declarations of new fixed point types, are not allowed.

15.a **Discussion:** This restriction would have the side effect of prohibiting the delay_relative_statement. As with the No_Floating_Point restriction, this might be used to avoid any question of rounding errors. Unless an Ada run-time is written in Ada, it seems hard to rule out implicit use of fixed point, since at the machine level, fixed point is virtually the same as integer arithmetic.

16/2 *This paragraph was deleted.*{*AI95-00394-01*}

17 No_Access_Subprograms
The declaration of access-to-subprogram types is not allowed.

17.a.1/2 **Discussion:** Most critical applications would require some restrictions or additional validation checks on uses of access-to-subprogram types. If the application does not require the functionality, then this restriction provides a means of ensuring the design requirement has been satisfied. The same applies to several of the following restrictions, and to restriction No_Dependence => Ada.Unchecked_Conversion.

18 No_Unchecked_Access
The Unchecked_Access attribute is not allowed.

19 No_Dispatch
Occurrences of T'Class are not allowed, for any (tagged) subtype T.

20/5 {*AI95-00285-01*} {*AI12-0318-1*} No_IO
Semantic dependence on any of the library units Sequential_IO, Direct_IO, Text_IO, Wide_Text_IO, Wide_Wide_Text_IO, Stream_IO, or Directories is not allowed.

20.a **Discussion:** Excluding the input-output facilities of an implementation may be needed in those environments which cannot support the supplied functionality. A program in such an environment is likely to require some low level facilities or a call on a non-Ada feature.

21 No_Delay Delay_Statements and semantic dependence on package Calendar are not allowed.

21.a **Ramification:** This implies that delay_alternatives in a select_statement are prohibited.

21.b The purpose of this restriction is to avoid the need for timing facilities within the run-time system.

22 No_Recursion
As part of the execution of a subprogram, the same subprogram is not invoked.

23 No_Reentrancy
During the execution of a subprogram by a task, no other task invokes the same subprogram.

23.1/5 {*AI12-0079-3*} No_Unspecified_Globals
No library-level entity shall have a Global aspect of Unspecified, either explicitly or by default. No library-level entity shall have a Global'Class aspect of Unspecified, explicitly or by default, if it is used as part of a dispatching call.

23.a/5 **Ramification:** Global'Class need not be specified on an operation if there are no dispatching calls to the operation, or if all of the dispatching calls are covered by dispatching_operation_specifiers for operations with such calls (see H.7).

23.2/5 {*AI12-0079-3*} {*AI12-0380-1*} No_Hidden_Indirect_Globals
When within a context where an applicable global aspect is neither Unspecified nor **in out all**, any execution within such a context does neither of the following:

23.3/5 - Update (or return a writable reference to) a variable that is reachable via a sequence of zero or more dereferences of access-to-object values from a parameter of a visibly access-to-constant type, from a part of a non-access-type formal parameter of mode **in** (after any **overriding** – see H.7), or from a global that has mode **in** or is not within the applicable global variable set, unless the initial dereference is of a part of a formal parameter or global that is visibly of an access-to-variable type;

- Read (or return a readable reference to) a variable that is reachable via a sequence of zero or more dereferences of access-to-object values from a global that is not within the applicable global variable set, unless the initial dereference is of a part of a formal parameter or global that is visibly of an access-to-object type. 23.4/5

Ramification: The above two rules specify that any hidden indirect references are covered by the global or formal parameter modes that apply, and are *not* subject to alternative paths of access (such as aliasing) that could result in conflicts. On the other hand, any visible access-to-object parts are allowed to designate objects that are accessible via other means, and side-effects on such objects are permitted if the value is visibly of an access-to-variable type. Such effects do not need to be covered by the applicable global aspect(s), but are rather for the caller to worry about. 23.b/5

For the purposes of the above rules: 23.5/5

- a part of an object is *visibly of an access type* if the type of the object is declared immediately within the visible part of a package specification, and at the point of declaration of the type the part is visible and of an access type; 23.6/5
- a function *returns a writable reference to V* if it returns a result with a part that is visibly of an access-to-variable type designating *V*; similarly, a function *returns a readable reference to V* if it returns a result with a part that is visibly of an access-to-constant type designating *V*; 23.7/5
- if an applicable global variable set includes a package name, and the collection of some pool-specific access type (see 7.6.1) is implicitly declared in a part of the declarative region of the package included within the global variable set, then all objects allocated from that collection are considered included within the global variable set. 23.8/5

The consequences of violating the No_Hidden_Indirect_Globals restriction is implementation-defined. Any aspects or other means for identifying such violations prior to or during execution are implementation-defined. 23.9/5

Implementation defined: The consequences of violating No_Hidden_Indirect_Globals. 23.b.1/5

Discussion: We do not make violations automatically erroneous, because if the implementation chooses to never fully trust it, there is nothing erroneous that can happen. If an implementation chooses to trust the restriction, and performs some optimization as a result of the restriction, the implementation would define such a violation as erroneous. Such an implementation might also endeavor to detect most violations, perhaps by providing additional aspects, thereby reducing the situations which result in erroneous execution. Implementations might detect some but not all violations of the restrictions. Implementations that completely ignore the restriction should treat the restriction as an unsupported capability of Annex H, “High Integrity Systems”. 23.c/5

Dynamic Semantics

{*AI12-0020-1*} {*AI12-0340-1*} The following *restriction_parameter*_identifier is language defined: 23.10/5

Max_Image_Length 23.11/5

Specifies the maximum length for the result of an Image, Wide_Image, or Wide_Wide_Image attribute. Violation of this restriction results in the raising of Program_Error at the point of the invocation of an image attribute.

Implementation Requirements

{*AI95-00394-01*} An implementation of this Annex shall support: 23.12/5

- the restrictions defined in this subclause; and 23.13/5
- {*AI05-0189-1*} the following restrictions defined in D.7: No_Task_Hierarchy, No_Abort_Statement, No_Implicit_Heap_Allocation, No_Standard_Allocators_After_Elaboration; and 23.14/5
- {*AI95-00347-01*} the **pragma** Profile(Ravenscar); and 23.15/5

23.d/2 **Discussion:** {*AI95-00347-01*} The reference to pragma Profile(Ravenscar) is intended to show that properly restricted tasking is appropriate for use in high integrity systems. The Ada 95 Annex seemed to suggest that tasking was inappropriate for such systems.

23.16/5 - the following uses of *restriction_parameter_*identifiers defined in D.7[, which are checked prior to program execution]:

23.17/5 - Max_Task_Entries => 0,

23.18/5 - Max_Asynchronous_Select_Nesting => 0, and

23.19/5 - Max_Tasks => 0.

23.20/5 {*AI12-0020-1*} {*AI12-0340-1*} If a Max_Image_Length restriction applies to any compilation unit in the partition, then for any subtype S, S'Image, S'Wide_Image, and S'Wide_Wide_Image shall be implemented within that partition without any dynamic allocation.

23.e/5 **Implementation Note:** {*AI12-0340-1*} {*AI12-0384-2*} This can be accomplished by using an object of the Text_Buffers.Bounded.Buffer_Type with the maximum characters as specified in the Max_Image_Length restriction, with a raise of Program_Error afterward if Text_Truncated (Buf) is True after the call on Put_Image (Buf, Arg).

24/5 {*AI05-0263-1*} {*AI05-0272-1*} {*AI12-0308-1*} If an implementation supports pragma Restrictions for a particular argument, then except for the restrictions No_Access_Subprograms, No_Unchecked_Access, No_Specification_of_Aspect, No_Use_of_Attribute, No_Use_of_Pragma, No_Dependence => Ada.Unchecked_Conversion, and No_Dependence => Ada.Unchecked_Deallocation, the associated restriction applies to the run-time system.

24.a **Reason:** Permission is granted for the run-time system to use the specified otherwise-restricted features, since the use of these features may simplify the run-time system by allowing more of it to be written in Ada.

24.b **Discussion:** The restrictions that are applied to the partition are also applied to the run-time system. For example, if No_Floating_Point is specified, then an implementation that uses floating point for implementing the delay statement (say) would require that No_Floating_Point is only used in conjunction with No_Delay. It is clearly important that restrictions are effective so that Max_Tasks=0 does imply that tasking is not used, even implicitly (for input-output, say).

24.c An implementation of tasking could be produced based upon a run-time system written in Ada in which the rendezvous was controlled by protected types. In this case, No_Protected_Types could only be used in conjunction with Max_Task_Entries=0. Other implementation dependencies could be envisaged.

24.d If the run-time system is not written in Ada, then the wording needs to be applied in an appropriate fashion.

Documentation Requirements

25 If a pragma Restrictions(No_Exceptions) is specified, the implementation shall document the effects of all constructs where language-defined checks are still performed automatically (for example, an overflow check performed by the processor).

25.a/2 *This paragraph was deleted.*

25.b/2 **Documentation Requirement:** If a pragma Restrictions(No_Exceptions) is specified, the effects of all constructs where language-defined checks are still performed.

25.c/2 **Discussion:** {*AI95-00114-01*} The documentation requirements here are quite difficult to satisfy. One method is to review the object code generated and determine the checks that are still present, either explicitly, or implicitly within the architecture. As another example from that of overflow, consider the question of dereferencing a null pointer. This could be undertaken by a memory access trap when checks are performed. When checks are suppressed via the argument No_Exceptions, it would not be necessary to have the memory access trap mechanism enabled.

Erroneous Execution

26 Program execution is erroneous if pragma Restrictions(No_Exceptions) has been specified and the conditions arise under which a generated language-defined runtime check would fail.

26.a **Discussion:** The situation here is very similar to the application of pragma Suppress. Since users are removing some of the protection the language provides, they had better be careful!

Program execution is erroneous if pragma Restrictions(No_Recursion) has been specified and a subprogram is invoked as part of its own execution, or if pragma Restrictions(No_Reentrancy) has been specified and during the execution of a subprogram by a task, another task invokes the same subprogram. 27

Discussion: {*AI05-0005-1*} In practice, many implementations might not exploit the absence of recursion or need for reentrancy, in which case the program execution would be unaffected by the use of recursion or reentrancy, even though the program is still formally erroneous. 27.a/3

This paragraph was deleted. 27.b/2

NOTE {*AI95-00394-01*} {*AI12-0440-1*} Uses of *restriction_parameter_*identifier No_Dependence defined in 13.12.1: No_Dependence => Ada.Unchecked_Deallocation and No_Dependence => Ada.Unchecked_Conversion can be appropriate for high-integrity systems. Other uses of No_Dependence can also be appropriate for high-integrity systems. 28/5

Discussion: The specific mention of these two uses is meant to replace the identifiers now banished to J.13, "Dependence Restriction Identifiers". 28.a/2

Restriction No_Dependence => Ada.Unchecked_Deallocation would be useful in those contexts in which heap storage is needed on program start-up, but need not be increased subsequently. The danger of a dangling pointer can therefore be avoided. 28.b/2

Extensions to Ada 95

{*8652/0042*} {*AI95-00130-01*} No_Local_Allocators no longer prohibits generic instantiations. 28.c/2

Wording Changes from Ada 95

{*AI95-00285-01*} Wide_Wide_Text_IO (which is new) is added to the No_IO restriction. 28.d/2

{*AI95-00347-01*} {*AI05-0299-1*} The title of this subclause was changed to match the change to the Annex title. Pragma Profile(Ravenscar) is part of this annex. 28.e/3

{*AI95-00394-01*} Restriction No_Dependence is used instead of special *restriction_*identifiers. The old names are banished to Obsolescent Features (see J.13). 28.f/2

{*AI95-00394-01*} The bizarre wording "apply in this Annex" (which no one quite can explain the meaning of) is banished. 28.g/2

Extensions to Ada 2005

{*AI05-0152-1*} {*AI05-0190-1*} Restrictions No_Anonymous_Allocators, No_Coextensions, and No_Access_Parameter_Allocators are new. 28.h/3

Wording Changes from Ada 2005

{*AI05-0189-1*} New restriction No_Standard_Allocators_After_Elaboration is added to the list of restrictions that are required by this annex. 28.i/3

{*AI05-0263-1*} **Correction:** Ada 2005 restriction No_Dependence is added where needed (this was missed in Ada 2005). 28.j/3

{*AI05-0272-1*} Restrictions against individual aspects, pragmas, and attributes do not apply to the run-time system, in order that an implementation can use whatever aspects, pragmas, and attributes are needed to do the job. For instance, attempting to write a run-time system for Linux that does not use the Import aspect would be very difficult and probably is not what the user is trying to prevent anyway. 28.k/3

Incompatibilities With Ada 2012

{*AI12-0318-1*} **Correction:** Restriction No_IO now excludes use of Ada.Directories. If a program using No_IO used Ada.Directories, it would be legal in Ada 2012 and illegal in Ada 2022. However, given the role of Ada.Directories as a support package for the other packages that are excluded by No_IO, it seems unlikely that any use of the restriction would use this package (and it's possible that implementations wouldn't support its use with No_IO anyway). 28.l/5

Extensions to Ada 2012

{*AI12-0020-1*} Restriction Max_Image_Length is new. 28.m/5

{*AI12-0079-3*} Restrictions No_Unspecified_Globals and No_Hidden_Indirect_Globals are new. 28.n/5

H.4.1 Aspect No_Controlled_Parts

Static Semantics

1/5 {*AI12-0256-1*} {*AI12-0403-1*} For a type, the following type-related, operational aspect may be specified:

2/5 No_Controlled_Parts

The type of this aspect is Boolean. If True, the type and any descendants shall not have any controlled parts. If specified, the value of the expression shall be static. If not specified, the value of this aspect is False.

2.a/5 **Aspect Description for No_Controlled_Parts:** A specification that a type and its descendants do not have controlled parts.

3/5 {*AI12-0256-1*} The No_Controlled_Parts aspect is nonoverridable (see 13.1.1).

3.a/5 **Discussion:** {*AI12-0407-1*} Since this is a Boolean-valued aspect, the blanket restrictions defined by 13.1.1 apply to the specification of Boolean-valued aspects on descendants of types with such aspects. But we still need rules about inheritance from progenitors and about hiding the aspect; it would be too painful to repeat those rules here (and have future maintenance fixes not get applied to this aspect).

Legality Rules

4/5 {*AI12-0256-1*} {*AI12-0407-1*} If No_Controlled_Parts is True for a type, no component of the type shall have a controlled part nor shall the type itself be controlled. For the purposes of this rule, a type has a controlled part if its full type has a controlled part; this is applied recursively. In addition to the places where Legality Rules normally apply (see 12.3), this rule also applies in the private part of an instance of a generic unit.

4.a/5 **Discussion:** {*AI12-0407-1*} This check breaks privacy by looking at the full definition of all of the types involved. This is more like a representation aspect than an operational aspect, but representation aspects are not allowed on partial views and we need this aspect to be visible to clients.

5/5 {*AI12-0256-1*} {*AI12-0407-1*} When enforcing the above rule within a generic body *G* or within the body of a generic unit declared within the declarative region of generic unit *G*, a generic formal private type of *G* and a generic formal derived type of *G* whose ancestor is a tagged type whose No_Controlled_Parts aspect is False are considered to have a controlled part.

5.a/5 **Reason:** This is a typical generic assume-the-worst rule.

5.b/5 **To be honest:** {*AI12-0407-1*} If the ancestor of the generic derived type is class-wide, the aspect in question belongs to the specific type associated with the class-wide type.

Extensions to Ada 2012

5.c/5 {*AI12-0256-1*} {*AI12-0407-1*} Aspect No_Controlled_Parts is new.

H.5 Pragma Detect_Blocking

1/5 {*AI95-00305-01*} {*AI12-0267-1*} The following pragma requires an implementation to detect potentially blocking operations during the execution of a protected operation or a parallel construct.

Syntax

2/2 {*AI95-00305-01*} The form of a pragma Detect_Blocking is as follows:

3/2 **pragma** Detect_Blocking;

Post-Compilation Rules

4/2 {*AI95-00305-01*} A pragma Detect_Blocking is a configuration pragma.

Dynamic Semantics

{*AI95-00305-01*} {*AI12-0247-1*} {*AI12-0267-1*} An implementation is required to detect a potentially blocking operation that occurs during the execution of a protected operation or a parallel construct defined within a compilation unit to which the pragma applies, and to raise Program_Error (see 9.5). 5/5

Implementation Permissions

{*AI95-00305-01*} {*AI12-0267-1*} An implementation is allowed to reject a compilation_unit to which a pragma Detect_Blocking applies if a potentially blocking operation is present directly within an entry_body, the body of a protected subprogram, or a parallel construct occurring within the compilation unit. 6/5

NOTE {*AI95-00305-01*} {*AI12-0442-1*} An operation that causes a task to be blocked within a foreign language domain is not defined to be potentially blocking, and is unlikely to be detected. 7/5

Extensions to Ada 95

{*AI95-00305-01*} Pragma Detect_Blocking is new. 7.a/2

Extensions to Ada 2012

{*AI12-0267-1*} Pragma Detect_Blocking now applies to parallel constructs as well as protected actions. 7.b/5

H.6 Pragma Partition_Elaboration_Policy

{*AI95-00265-01*} {*AI05-0299-1*} This subclause defines a pragma for user control over elaboration policy. 1/3

Syntax

{*AI95-00265-01*} The form of a pragma Partition_Elaboration_Policy is as follows: 2/2

pragma Partition_Elaboration_Policy (*policy*_identifier); 3/2

The *policy*_identifier shall be either Sequential, Concurrent or an implementation-defined identifier. 4/2

Implementation defined: Implementation-defined *policy*_identifiers allowed in a pragma Partition_Elaboration_Policy. 4.a/2

Ramification: Note that the Ravenscar profile (see D.13) has nothing to say about which Partition_Elaboration_Policy is used. This was intentionally omitted from the profile, as there was no agreement as to whether the Sequential policy should be required for Ravenscar programs. As such it was defined separately. 4.b/3

Post-Compilation Rules

{*AI95-00265-01*} A pragma Partition_Elaboration_Policy is a configuration pragma. It specifies the elaboration policy for a partition. At most one elaboration policy shall be specified for a partition. 5/2

{*AI95-00265-01*} {*AI05-0264-1*} If the Sequential policy is specified for a partition, then pragma Restrictions (No_Task_Hierarchy) shall also be specified for the partition. 6/3

Dynamic Semantics

{*AI95-00265-01*} Notwithstanding what this document says elsewhere, this pragma allows partition elaboration rules concerning task activation and interrupt attachment to be changed. If the *policy*_identifier is Concurrent, or if there is no pragma Partition_Elaboration_Policy defined for the partition, then the rules defined elsewhere in this Reference Manual apply. 7/2

{*AI95-00265-01*} {*AI95-00421-01*} If the partition elaboration policy is Sequential, then task activation and interrupt attachment are performed in the following sequence of steps: 8/2

9/2 - The activation of all library-level tasks and the attachment of interrupt handlers are deferred until all library units are elaborated.

10/2 - The interrupt handlers are attached by the environment task.

11/2 - The environment task is suspended while the library-level tasks are activated.

12/2 - The environment task executes the main subprogram (if any) concurrently with these executing tasks.

13/2 {*AI95-00265-01*} {*AI95-00421-01*} If several dynamic interrupt handler attachments for the same interrupt are deferred, then the most recent call of Attach_Handler or Exchange_Handler determines which handler is attached.

14/2 {*AI95-00265-01*} {*AI95-00421-01*} If any deferred task activation fails, Tasking_Error is raised at the beginning of the sequence of statements of the body of the environment task prior to calling the main subprogram.

Implementation Advice

15/3 {*AI95-00265-01*} {*AI05-0264-1*} If the partition elaboration policy is Sequential and the Environment task becomes permanently blocked during elaboration, then the partition is deadlocked and it is recommended that the partition be immediately terminated.

15.a/3 **Implementation Advice:** If the partition elaboration policy is Sequential and the Environment task becomes permanently blocked during elaboration, then the partition should be immediately terminated.

Implementation Permissions

16/3 {*AI95-00265-01*} {*AI05-0264-1*} If the partition elaboration policy is Sequential and any task activation fails, then an implementation may immediately terminate the active partition to mitigate the hazard posed by continuing to execute with a subset of the tasks being active.

17/5 NOTE {*AI95-00421-01*} {*AI12-0440-1*} If any deferred task activation fails, the environment task is unable to handle the Tasking_Error exception and completes immediately. By contrast, if the partition elaboration policy is Concurrent, then this exception can be handled within a library unit.

Extensions to Ada 95

17.a/2 {*AI95-00265-01*} {*AI95-00421-01*} Pragma Partition_Elaboration_Policy is new.

H.7 Extensions to Global and Global'Class Aspects

1/5 {*AI12-0079-3*} In addition to the entities specified in 6.1.2, the Global aspect may be specified for a subtype (including a formal subtype), formal package, formal subprogram, and formal object of an anonymous access-to-subprogram type.

Syntax

2/5 {*AI12-0079-3*} {*AI12-0380-1*} The following additional syntax is provided to override the mode of a formal parameter to reflect indirect effects on variables reachable from the formal parameter by one or more access-value dereferences:

3/5
```
extended_global_mode ::=
   overriding basic_global_mode
```

Name Resolution Rules

4/5 {*AI12-0079-3*} The *object_*name that is associated with an **overriding** mode shall resolve to statically denote a formal object, or a formal parameter of the associated entity.

Static Semantics

{*AI12-0079-3*} The presence of the reserved word **overriding** in a global mode indicates that the specification is overriding the mode of a formal parameter with another mode to reflect the overall effect of an invocation of the callable entity on the state associated with the corresponding actual parameter. 5/5

{*AI12-0380-1*} [As described in 6.1.2, the following rules are defined in terms of operations that are performed by or on behalf of an entity.] 6/5

{*AI12-0079-3*} {*AI12-0431-1*} {*AI12-0439-1*} The Global aspect for a subtype identifies the global variables that can be referenced during default initialization, adjustment as part of assignment, finalization of an object of the subtype, or conversion to the subtype, including the evaluation of any assertion expressions that apply. If not specified for the first subtype of a derived type, the aspect defaults to that of the ancestor subtype; if not specified for a nonderived composite first subtype the aspect defaults to that of the enclosing library unit; if not specified for a nonderived elementary first subtype (or scalar base subtype), the aspect defaults to **null** in the absence of a predicate (or when the predicate is statically True), and to that of the enclosing library unit otherwise. If not specified for a nonfirst subtype *S*, the Global aspect defaults to that of the subtype identified in the subtype_indication defining *S*. 7/5

{*AI12-0079-3*} The Global'Class aspect may be specified for the first subtype of a tagged type *T*, indicating an upper bound on the Global aspect of any descendant of *T*. If not specified, it defaults to Unspecified. 8/5

Legality Rules

{*AI12-0079-3*} For a tagged subtype *T*, each mode of its Global aspect shall identify a subset of the variables identified either by the corresponding mode, or by the **in out** mode, of the Global'Class aspect of the first subtype of any ancestor of *T*. 9/5

Extensions to Ada 2012

{*AI12-0079-3*} {*AI12-0380-1*} These extensions to the Global aspect are new. 9.a/5

H.7.1 The Use_Formal and Dispatching Aspects

{*AI12-0380-1*} The Use_Formal and Dispatching aspects are provided to more precisely describe the use of generic formal parameters and dispatching calls within the execution of an operation, enabling more precise checking of conformance with the Nonblocking and global aspects that apply at the point of invocation of the operation. 1/5

{*AI12-0079-3*} {*AI12-0380-1*} For any declaration within a generic unit for which a global or Nonblocking aspect may be specified, other than a generic_formal_parameter_declaration, the following aspect may be specified to indicate which generic formal parameters are *used* by the associated entity: 2/5

{*AI12-0380-1*} Use_Formal 3/5

The aspect is specified with a formal_parameter_set, with the following form:

Aspect Description for Use_Formal: Generic formal parameters used in the implementation of an entity. 3.a/5

```
formal_parameter_set ::=                                                    4/5
   formal_group_designator
 | formal_parameter_name
 | (formal_parameter_name{, formal_parameter_name})

formal_group_designator ::= null | all                                      5/5

formal_parameter_name ::=                                                   6/5
   formal_subtype_mark
```

| *formal_subprogram_*name
| *formal_access_to_subprogram_object_*name

7/5 {*AI12-0079-3*} {*AI12-0380-1*} For any declaration for which a global or Nonblocking aspect may be specified, other than for a library package, a generic library package, or a generic formal, the following aspect may be specified:

8/5 {*AI12-0380-1*} Dispatching

The aspect is specified with a dispatching_operation_set, with the following form:

8.a/5 **Aspect Description for Dispatching:** Generic formal parameters used in the implementation of an entity.

9/5 dispatching_operation_set ::=
dispatching_operation_specifier
| (dispatching_operation_specifier{, dispatching_operation_specifier})

10/5 dispatching_operation_specifier ::=
*dispatching_operation_*name (*object_*name)

Name Resolution Rules

11/5 {*AI12-0380-1*} A formal_parameter_name in a Use_Formal aspect shall resolve to statically denote a formal subtype, a formal subprogram, or a formal object of an anonymous access-to-subprogram type[of an enclosing generic unit or visible formal package].

12/5 {*AI12-0380-1*} The *object_*name of a dispatching_operation_specifier shall resolve to statically name an object (including possibly a formal parameter) of a tagged class-wide type *T*'Class, or of an access type designating a tagged class-wide type *T*'Class; the *dispatching_operation_*name of the dispatching_operation_specifier shall resolve to statically denote a dispatching operation associated with *T*.

Static Semantics

13/5 {*AI12-0380-1*} {*AI12-0439-1*} The *formal parameter set* is identified by a set of formal_parameter_names. Alternatively, the reserved word **null** may be used to indicate none of the generic formal parameters, or **all** to indicate all of the generic formal parameters, of any enclosing generic unit (or visible formal package) can be used within the execution of the operation. If there is no formal parameter set specified for an entity declared within a generic unit, it defaults to **all**.

14/5 {*AI12-0380-1*} {*AI12-0404-1*} {*AI12-0444-1*} The *dispatching operation set* is identified by a set of dispatching_operation_specifiers. It indicates that the Nonblocking and global effects of dispatching calls that match one of the specifiers, rather than being accounted for by the Nonblocking or global aspect, are instead to be accounted for by the invoker of the operation. A dispatching call matches a dispatching_operation_specifier if the name or prefix of the call statically denotes the same operation(s) as that of the dispatching_operation_specifier, and at least one of the objects controlling the call is denoted by, or designated by, a name that statically names the same object as that denoted by the *object_*name of the dispatching_operation_specifier.

14.a/5 **Ramification:** The object "controlling the call" is not necessarily a controlling parameter of the call if the call is a function with a controlling result or has parameters that is such a function. It is one of the objects that provide the dispatching tag used for the call; that could, for example, be a parameter of a function used as a parameter to the call, or an object being assigned to, or a parameter of an enclosing call.

15/5 {*AI12-0380-1*} {*AI12-0404-1*} In the absence of any dispatching_operation_specifiers, or if none of them match a dispatching call *C* within an operation *P*, Nonblocking and global aspects checks are performed at the point of the call *C* within *P* using the Nonblocking and Global'Class aspects that apply to the dispatching operation named in call *C*. If there is a match, any global access or potential blocking within the subprogram body invoked by the call *C* is ignored at the point of call within *P*. Instead, when

the operation *P* itself is invoked, Nonblocking and global aspect checks are performed presuming each named dispatching operation is called at least once (with the named object controlling the call), but similarly ignoring those dispatching calls that would match a dispatching_operation_specifier applicable at the point of invocation of *P*.

Legality Rules

{*AI12-0380-1*} Within an operation to which a Use_Formal aspect applies, if the formal parameter set is anything but **all**, then the only generic formal subtypes that may be used, the only formal subprograms that may be called, and the only formal objects of an anonymous access-to-subprogram type that may be dereferenced as part of a call or passed as the actual for an access parameter, are those included in the formal parameter set. 16/5

{*AI12-0380-1*} When an operation (or instance thereof) to which a Use_Formal aspect applies is invoked, Nonblocking and global aspect checks are performed presuming each generic formal parameter (or corresponding actual parameter) of the formal parameter set is used at least once. 17/5

Examples

{*AI12-0430-1*} *An example of use of the Dispatching aspect:* 18/5

```
procedure My_Write(  --  see 13.13.2                                    19/5
   Stream : not null access Ada.Streams.Root_Stream_Type'Class;
   Item   : My_Integer'Base)
   with Dispatching => Write(Stream);
for My_Integer'Write use My_Write;
```

{*AI12-0430-1*} *For examples of use of the Use_Formal aspect, see the Element functions of Hashed_Sets in A.18.8.* 20/5

Extensions to Ada 2012

{*AI12-0079-3*} The aspects Use_Formal and Dispatching are new. 20.a/5

Annex J
(normative)
Obsolescent Features

{*AI95-00368-01*} [This Annex contains descriptions of features of the language whose functionality is largely redundant with other features defined by this Reference Manual. Use of these features is not recommended in newly written programs. Use of these features can be prevented by using pragma Restrictions (No_Obsolescent_Features), see 13.12.1.] 1/2

Ramification: These features are still part of the language, and have to be implemented by conforming implementations. The primary reason for putting these descriptions here is to get redundant features out of the way of most readers. The designers of the next version of Ada will have to assess whether or not it makes sense to drop these features from the language. 1.a

Wording Changes from Ada 83

The following features have been removed from the language, rather than declared to be obsolescent: 1.b

- The package Low_Level_IO (see A.6). 1.c
- The Epsilon, Mantissa, Emax, Small, Large, Safe_Emax, Safe_Small, and Safe_Large attributes of floating point types (see A.5.3). 1.d
- *This paragraph was deleted.*{*AI95-00284-02*} 1.e/2
- The pragmas System_Name, Storage_Unit, and Memory_Size (see 13.7). 1.f
- The pragma Shared (see C.6). 1.g

Implementations can continue to support the above features for upward compatibility. 1.h

Wording Changes from Ada 95

{*AI95-00368-01*} A mention of the No_Obsolescent_Features restriction was added. 1.i/2

Wording Changes from Ada 2005

{*AI05-0229-1*} Pragma Controlled has been removed from the language, rather than declared to be obsolescent. No existing implementation gives it any effect. An implementation could continue to support the pragma as an implementation-defined pragma for upward compatibility. 1.j/3

J.1 Renamings of Library Units

Static Semantics

The following library_unit_renaming_declarations exist: 1

```
with Ada.Unchecked_Conversion;                                                    2
generic function Unchecked_Conversion renames Ada.Unchecked_Conversion;

with Ada.Unchecked_Deallocation;                                                  3
generic procedure Unchecked_Deallocation renames Ada.Unchecked_Deallocation;

with Ada.Sequential_IO;                                                           4
generic package Sequential_IO renames Ada.Sequential_IO;

with Ada.Direct_IO;                                                               5
generic package Direct_IO renames Ada.Direct_IO;

with Ada.Text_IO;                                                                 6
package Text_IO renames Ada.Text_IO;

with Ada.IO_Exceptions;                                                           7
package IO_Exceptions renames Ada.IO_Exceptions;

with Ada.Calendar;                                                                8
package Calendar renames Ada.Calendar;
```

9
```
with System.Machine_Code;
package Machine_Code renames System.Machine_Code; -- If supported.
```

9.a/3 **Discussion:** {*AI05-0004-1*} These library units correspond to those declared in Ada 83, which did not have the child unit concept or the parent package Ada.

Implementation Requirements

10 The implementation shall allow the user to replace these renamings.

J.2 Allowed Replacements of Characters

Syntax

1 The following replacements are allowed for the vertical line, number sign, and quotation mark characters:

2 • A vertical line character (|) can be replaced by an exclamation mark (!) where used as a delimiter.

3 • The number sign characters (#) of a based_literal can be replaced by colons (:) provided that the replacement is done for both occurrences.

3.a/2 **To be honest:** {*AI95-00285-01*} The intent is that such a replacement works in the Value, Wide_Value, and Wide_Wide_Value attributes, and in the Get procedures of Text_IO (and Wide_Text_IO and Wide_Wide_Text_IO as well)}, so that things like “16:.123:” is acceptable.

4 • The quotation marks (") used as string brackets at both ends of a string literal can be replaced by percent signs (%) provided that the enclosed sequence of characters contains no quotation mark, and provided that both string brackets are replaced. Any percent sign within the sequence of characters shall then be doubled and each such doubled percent sign is interpreted as a single percent sign character value.

5 These replacements do not change the meaning of the program.

5.a **Reason:** The original purpose of this feature was to support hardware (for example, teletype machines) that has long been obsolete. The feature is no longer necessary for that reason. Another use of the feature has been to replace the vertical line character (|) when using certain hardware that treats that character as a (non-English) letter. The feature is no longer necessary for that reason, either, since Ada 95 has full support for international character sets. Therefore, we believe this feature is no longer necessary.

5.b Users of equipment that still uses | to represent a letter will continue to do so. Perhaps by next the time Ada is revised, such equipment will no longer be in use.

5.c Note that it was never legal to use this feature as a convenient method of including double quotes in a string without doubling them — the string literal:

5.d
```
%"This is quoted."%
```

5.e/3 {*AI05-0248-1*} is not legal in Ada (and never was legal). One has to write:

5.f
```
"""This is quoted."""
```

J.3 Reduced Accuracy Subtypes

1 A digits_constraint may be used to define a floating point subtype with a new value for its requested decimal precision, as reflected by its Digits attribute. Similarly, a delta_constraint may be used to define an ordinary fixed point subtype with a new value for its *delta*, as reflected by its Delta attribute.

1.a **Discussion:** It might be more direct to make these attributes specifiable via an attribute_definition_clause, and eliminate the syntax for these _constraints.

Syntax

2/4 {*AI12-0152-1*} delta_constraint ::= **delta** *static*_simple_expression [range_constraint]

Name Resolution Rules

{*AI12-0152-1*} The simple_expression of a delta_constraint is expected to be of any real type. 3/4

Legality Rules

{*AI12-0152-1*} The simple_expression of a delta_constraint shall be static. 4/4

For a subtype_indication with a delta_constraint, the subtype_mark shall denote an ordinary fixed point subtype. 5

For a subtype_indication with a digits_constraint, the subtype_mark shall denote either a decimal fixed point subtype or a floating point subtype (notwithstanding the rule given in 3.5.9 that only allows a decimal fixed point subtype). 6

This paragraph was deleted.{*AI95-00114-01*} 6.a/2

Static Semantics

{*AI12-0152-1*} A subtype_indication with a subtype_mark that denotes an ordinary fixed point subtype and a delta_constraint defines an ordinary fixed point subtype with a *delta* given by the value of the simple_expression of the delta_constraint. If the delta_constraint includes a range_constraint, then the ordinary fixed point subtype is constrained by the range_constraint. 7/4

{*AI12-0152-1*} A subtype_indication with a subtype_mark that denotes a floating point subtype and a digits_constraint defines a floating point subtype with a requested decimal precision (as reflected by its Digits attribute) given by the value of the simple_expression of the digits_constraint. If the digits_constraint includes a range_constraint, then the floating point subtype is constrained by the range_constraint. 8/4

Dynamic Semantics

{*AI12-0152-1*} A delta_constraint is *compatible* with an ordinary fixed point subtype if the value of the simple_expression is no less than the *delta* of the subtype, and the range_constraint, if any, is compatible with the subtype. 9/4

{*AI12-0152-1*} A digits_constraint is *compatible* with a floating point subtype if the value of the simple_expression is no greater than the requested decimal precision of the subtype, and the range_constraint, if any, is compatible with the subtype. 10/4

The elaboration of a delta_constraint consists of the elaboration of the range_constraint, if any. 11

Reason: A numeric subtype is considered “constrained” only if a range constraint applies to it. The only effect of a digits_constraint or a delta_constraint without a range_constraint is to specify the value of the corresponding Digits or Delta attribute in the new subtype. The set of values of the subtype is not “constrained” in any way by such _constraints. 11.a

Wording Changes from Ada 83

In Ada 83, a delta_constraint is called a fixed_point_constraint, and a digits_constraint is called a floating_point_constraint. We have adopted other terms because digits_constraints apply primarily to decimal fixed point types now (they apply to floating point types only as an obsolescent feature). 11.b

Wording Changes from Ada 2012

{*AI12-0152-1*} **Corrigendum:** Changed the syntax so that the value following **delta** in a delta_constraint is a simple_expression. This is compatible as any expressions that would require extra parentheses are already illegal. The change is necessary to eliminate syntax ambguities in derived_type_definitions. The similar change for digits_constraint is documented in 3.5.9. 11.c/4

J.4 The Constrained Attribute

Static Semantics

1 For every private subtype S, the following attribute is defined:

1.a **Discussion:** This includes generic formal private subtypes.

2 S'Constrained

Yields the value False if S denotes an unconstrained nonformal private subtype with discriminants; also yields the value False if S denotes a generic formal private subtype, and the associated actual subtype is either an unconstrained subtype with discriminants or an unconstrained array subtype; yields the value True otherwise. The value of this attribute is of the predefined subtype Boolean.

2.a **Reason:** Because Ada 95 has unknown_discriminant_parts, the Constrained attribute of private subtypes is obsolete. This is fortunate, since its Ada 83 definition was confusing, as explained below. Because this attribute is obsolete, we do not bother to extend its definition to private extensions.

2.b The Constrained attribute of an object is *not* obsolete.

2.c Note well: S'Constrained matches the Ada 95 definition of "constrained" only for composite subtypes. For elementary subtypes, S'Constrained is always true, whether or not S is constrained. (The Constrained attribute of an object does not have this problem, as it is only defined for objects of a discriminated type.) So one should think of its designator as being 'Constrained_Or_Elementary.

J.5 ASCII

Static Semantics

1 The following declaration exists in the declaration of package Standard:

2
```
package ASCII is
```

3
```
   -- Control characters:
```

4
```
   NUL    : constant Character := nul;    SOH    : constant Character := soh;
   STX    : constant Character := stx;    ETX    : constant Character := etx;
   EOT    : constant Character := eot;    ENQ    : constant Character := enq;
   ACK    : constant Character := ack;    BEL    : constant Character := bel;
   BS     : constant Character := bs;     HT     : constant Character := ht;
   LF     : constant Character := lf;     VT     : constant Character := vt;
   FF     : constant Character := ff;     CR     : constant Character := cr;
   SO     : constant Character := so;     SI     : constant Character := si;
   DLE    : constant Character := dle;    DC1    : constant Character := dc1;
   DC2    : constant Character := dc2;    DC3    : constant Character := dc3;
   DC4    : constant Character := dc4;    NAK    : constant Character := nak;
   SYN    : constant Character := syn;    ETB    : constant Character := etb;
   CAN    : constant Character := can;    EM     : constant Character := em;
   SUB    : constant Character := sub;    ESC    : constant Character := esc;
   FS     : constant Character := fs;     GS     : constant Character := gs;
   RS     : constant Character := rs;     US     : constant Character := us;
   DEL    : constant Character := del;
```

5
```
   -- Other characters:
```

6
```
   Exclam    : constant Character:= '!'; Quotation : constant Character:= '"';
   Sharp     : constant Character:= '#'; Dollar    : constant Character:= '$';
   Percent   : constant Character:= '%'; Ampersand : constant Character:= '&';
   Colon     : constant Character:= ':'; Semicolon : constant Character:= ';';
   Query     : constant Character:= '?'; At_Sign   : constant Character:= '@';
   L_Bracket: constant Character:= '['; Back_Slash: constant Character:= '\';
   R_Bracket: constant Character:= ']'; Circumflex: constant Character:= '^';
   Underline: constant Character:= '_'; Grave     : constant Character:= '`';
   L_Brace   : constant Character:= '{'; Bar       : constant Character:= '|';
   R_Brace   : constant Character:= '}'; Tilde     : constant Character:= '~';
```

```
   -- Lower case letters:
   LC_A: constant Character:= 'a';
   ...
   LC_Z: constant Character:= 'z';
end ASCII;
```

J.6 Numeric_Error

Static Semantics

The following declaration exists in the declaration of package Standard:

```
Numeric_Error : exception renames Constraint_Error;
```

Discussion: This is true even though it is not shown in A.1.

Reason: In Ada 83, it was unclear which situations should raise Numeric_Error, and which should raise Constraint_Error. The permissions of RM83-11.6 could often be used to allow the implementation to raise Constraint_Error in a situation where one would normally expect Numeric_Error. To avoid this confusion, all situations that raise Numeric_Error in Ada 83 are changed to raise Constraint_Error in Ada 95. Numeric_Error is changed to be a renaming of Constraint_Error to avoid most of the upward compatibilities associated with this change.

In new code, Constraint_Error should be used instead of Numeric_Error.

J.7 At Clauses

Syntax

at_clause ::= **for** direct_name **use at** expression;

Static Semantics

An at_clause of the form "for *x* use at *y*;" is equivalent to an attribute_definition_clause of the form "for *x*'Address use *y*;".

Reason: The preferred syntax for specifying the address of an entity is an attribute_definition_clause specifying the Address attribute. Therefore, the special-purpose at_clause syntax is now obsolete.

The above equivalence implies, for example, that only one at_clause is allowed for a given entity. Similarly, it is illegal to give both an at_clause and an attribute_definition_clause specifying the Address attribute.

Extensions to Ada 83

We now allow to define the address of an entity using an attribute_definition_clause. This is because Ada 83's at_clause is so hard to remember: programmers often tend to write "for X'Address use...;".

Wording Changes from Ada 83

Ada 83's address_clause is now called an at_clause to avoid confusion with the new term "Address clause" (that is, an attribute_definition_clause for the Address attribute).

J.7.1 Interrupt Entries

[Implementations are permitted to allow the attachment of task entries to interrupts via the address clause. Such an entry is referred to as an *interrupt entry*.

The address of the task entry corresponds to a hardware interrupt in an implementation-defined manner. (See Ada.Interrupts.Reference in C.3.2.)]

Static Semantics

The following attribute is defined:

4 For any task entry X:

5 X'Address For a task entry whose address is specified (an *interrupt entry*), the value refers to the corresponding hardware interrupt. For such an entry, as for any other task entry, the meaning of this value is implementation defined. The value of this attribute is of the type of the subtype System.Address.

6 Address may be specified for single entries via an attribute_definition_clause.

6.a **Reason:** Because of the equivalence of at_clauses and attribute_definition_clauses, an interrupt entry may be specified via either notation.

Dynamic Semantics

7 As part of the initialization of a task object, the address clause for an interrupt entry is elaborated[, which evaluates the expression of the address clause]. A check is made that the address specified is associated with some interrupt to which a task entry may be attached. If this check fails, Program_Error is raised. Otherwise, the interrupt entry is attached to the interrupt associated with the specified address.

8 Upon finalization of the task object, the interrupt entry, if any, is detached from the corresponding interrupt and the default treatment is restored.

9 While an interrupt entry is attached to an interrupt, the interrupt is reserved (see C.3).

10 An interrupt delivered to a task entry acts as a call to the entry issued by a hardware task whose priority is in the System.Interrupt_Priority range. It is implementation defined whether the call is performed as an ordinary entry call, a timed entry call, or a conditional entry call; which kind of call is performed can depend on the specific interrupt.

Bounded (Run-Time) Errors

11 It is a bounded error to evaluate E'Caller (see C.7.1) in an accept_statement for an interrupt entry. The possible effects are the same as for calling Current_Task from an entry body.

Documentation Requirements

12 The implementation shall document to which interrupts a task entry may be attached.

12.a/2 **Documentation Requirement:** The interrupts to which a task entry may be attached.

13 The implementation shall document whether the invocation of an interrupt entry has the effect of an ordinary entry call, conditional call, or a timed call, and whether the effect varies in the presence of pending interrupts.

13.a/2 **Documentation Requirement:** The type of entry call invoked for an interrupt entry.

Implementation Permissions

14 The support for this subclause is optional.

15 Interrupts to which the implementation allows a task entry to be attached may be designated as reserved for the entire duration of program execution[; that is, not just when they have an interrupt entry attached to them].

16/1 {*8652/0077*} {*AI95-00111-01*} Interrupt entry calls may be implemented by having the hardware execute directly the appropriate accept_statement. Alternatively, the implementation is allowed to provide an internal interrupt handler to simulate the effect of a normal task calling the entry.

17 The implementation is allowed to impose restrictions on the specifications and bodies of tasks that have interrupt entries.

18 It is implementation defined whether direct calls (from the program) to interrupt entries are allowed.

If a select_statement contains both a terminate_alternative and an accept_alternative for an interrupt entry, then an implementation is allowed to impose further requirements for the selection of the terminate_alternative in addition to those given in 9.3. 19

NOTE 1 {*8652/0077*} {*AI95-00111-01*} Queued interrupts correspond to ordinary entry calls. Interrupts that are lost if not immediately processed correspond to conditional entry calls. It is a consequence of the priority rules that an accept_statement executed in response to an interrupt can be executed with the active priority at which the hardware generates the interrupt, taking precedence over lower priority tasks, without a scheduling action. 20/1

NOTE 2 Control information that is supplied upon an interrupt can be passed to an associated interrupt entry as one or more parameters of mode **in**. 21

Examples

Example of an interrupt entry: 22

```
task Interrupt_Handler is                                                    23
  entry Done;
  for Done'Address use Ada.Interrupts.Reference(Ada.Interrupts.Names.Device_Done);
end Interrupt_Handler;
```

Wording Changes from Ada 83

{*AI95-00114-01*} RM83-13.5.1 did not adequately address the problems associated with interrupts. This feature is now obsolescent and is replaced by the Ada 95 interrupt model as specified in the Systems Programming Annex. 23.a/2

Wording Changes from Ada 95

{*8652/0077*} {*AI95-00111-01*} **Corrigendum:** The undefined term *accept body* was replaced by accept_statement. 23.b/2

J.8 Mod Clauses

Syntax

mod_clause ::= **at mod** *static_*expression; 1

Static Semantics

A record_representation_clause of the form: 2

```
{AI05-0092-1} for r use                                                      3/3
    record at mod a;
        ...
    end record;
```

is equivalent to: 4

```
for r'Alignment use a;                                                       5
for r use
    record
        ...
    end record;
```

Reason: The preferred syntax for specifying the alignment of an entity is an attribute_definition_clause specifying the Alignment attribute. Therefore, the special-purpose mod_clause syntax is now obsolete. 5.a

The above equivalence implies, for example, that it is illegal to give both a mod_clause and an attribute_definition_clause specifying the Alignment attribute for the same type. 5.b

Wording Changes from Ada 83

Ada 83's alignment_clause is now called a mod_clause to avoid confusion with the new term “Alignment clause” (that is, an attribute_definition_clause for the Alignment attribute). 5.c

J.9 The Storage_Size Attribute

Static Semantics

1 For any task subtype T, the following attribute is defined:

2 T'Storage_Size

Denotes an implementation-defined value of type *universal_integer* representing the number of storage elements reserved for a task of the subtype T.

2.a/3 **To be honest:** {*AI05-0229-1*} T'Storage_Size cannot be particularly meaningful in the presence of the specification of the aspect Storage_Size, especially when the expression is dynamic, or depends on a discriminant of the task, because the Storage_Size will be different for different objects of the type. Even without such a specification, the Storage_Size can be different for different objects of the type, and in any case, the value is implementation defined. Hence, it is always implementation defined.

3/3 {*AI95-00345-01*} {*AI05-0229-1*} Storage_Size may be specified for a task first subtype that is not an interface via an attribute_definition_clause. When the attribute is specified, the Storage_Size aspect is specified to be the value of the given expression.

3.a/3 **Ramification:** {*AI05-0229-1*} When this attribute is specified with an attribute_definition_clause, the associated aspect is set to the *value* of the expression given in the attribute_definition_clause, rather than the expression itself. This value is therefore the same for all objects of the type; in particular, it is not re-evaluated when objects are created. This is different than when the aspect is specified with an aspect_specification (see 13.3).

Wording Changes from Ada 95

3.b/2 {*AI95-00345-01*} We don't allow specifying Storage_Size on task interfaces. We don't need to mention class-wide task types, because these cannot be a first subtype.

J.10 Specific Suppression of Checks

1/2 {*AI95-00224-01*} Pragma Suppress can be used to suppress checks on specific entities.

Syntax

2/2 {*AI95-00224-01*} The form of a specific Suppress pragma is as follows:

3/2 **pragma** Suppress(identifier, [On =>] name);

Legality Rules

4/2 {*AI95-00224-01*} The identifier shall be the name of a check (see 11.5). The name shall statically denote some entity.

5/2 {*AI95-00224-01*} For a specific Suppress pragma that is immediately within a package_specification, the name shall denote an entity (or several overloaded subprograms) declared immediately within the package_specification.

Static Semantics

6/2 {*AI95-00224-01*} A specific Suppress pragma applies to the named check from the place of the pragma to the end of the innermost enclosing declarative region, or, if the pragma is given in a package_specification, to the end of the scope of the named entity. The pragma applies only to the named entity, or, for a subtype, on objects and values of its type. A specific Suppress pragma suppresses the named check for any entities to which it applies (see 11.5). Which checks are associated with a specific entity is not defined by this document.

6.a/2 **Discussion:** The language doesn't specify exactly which entities control whether a check is performed. For example, in

6.b
```
pragma Suppress (Range_Check, On => A);
A := B;
```

whether or not the range check is performed is not specified. The compiler may require that checks are suppressed on B or on the type of A in order to omit the range check. 6.c

Implementation Permissions

{*AI95-00224-01*} An implementation is allowed to place restrictions on specific Suppress pragmas. 7/2

NOTE {*AI95-00224-01*} {*AI12-0440-1*} An implementation can support a similar On parameter on pragma Unsuppress (see 11.5). 8/5

Wording Changes from Ada 95

{*AI95-00224-01*} {*AI05-0299-1*} This subclause is new. This feature was moved here because it is important for pragma Unsuppress that there be an unambiguous meaning for each checking pragma. For instance, in the example 8.a/3

```
pragma Suppress (Range_Check);
pragma Unsuppress (Range_Check, On => A);
A := B;
```
8.b

the user needs to be able to depend on the range check being made on the assignment. But a compiler survey showed that the interpretation of this feature varied widely; trying to define this carefully was likely to cause a lot of user and implementer pain. Thus the feature was moved here, to emphasize that its use is not portable. 8.c

J.11 The Class Attribute of Untagged Incomplete Types

Static Semantics

{*AI95-00326-01*} For the first subtype S of a type *T* declared by an incomplete_type_declaration that is not tagged, the following attribute is defined: 1/2

{*AI95-00326-01*} S'Class 2/2

Denotes the first subtype of the incomplete class-wide type rooted at *T*. The completion of *T* shall declare a tagged type. Such an attribute reference shall occur in the same library unit as the incomplete_type_declaration.

Reason: {*AI95-00326-01*} This must occur in the same unit to prevent children from imposing requirements on their ancestor library units for deferred incomplete types. 2.a/2

Wording Changes from Ada 95

{*AI95-00326-01*} {*AI05-0299-1*} This subclause is new. This feature was moved here because the tagged incomplete type provides a better way to provide this capability (it doesn't put requirements on the completion based on uses that could be anywhere). Pity we didn't think of it in 1994. 2.b/3

J.12 Pragma Interface

Syntax

{*AI95-00284-02*} In addition to an identifier, the reserved word **interface** is allowed as a pragma name, to provide compatibility with a prior edition of this Reference Manual. 1/2

Implementation Note: {*AI95-00284-02*} All implementations need to at least recognize and ignore this pragma. A syntax error is not an acceptable implementation of this pragma. 1.a/2

Wording Changes from Ada 95

{*AI95-00326-01*} {*AI05-0299-1*} This subclause is new. This is necessary as **interface** is now a reserved word, which would prevent pragma Interface from being an implementation-defined pragma. We don't define any semantics for this pragma, as we expect that implementations will continue to use whatever they currently implement - requiring any changes would be counter-productive. 1.b/3

J.13 Dependence Restriction Identifiers

1/2 {*AI95-00394-01*} The following restrictions involve dependence on specific language-defined units. The more general restriction No_Dependence (see 13.12.1) should be used for this purpose.

Static Semantics

2/2 {*AI95-00394-01*} The following *restriction*_identifiers exist:

3/2 {*AI95-00394-01*} No_Asynchronous_Control
Semantic dependence on the predefined package Asynchronous_Task_Control is not allowed.

4/2 {*AI95-00394-01*} No_Unchecked_Conversion
Semantic dependence on the predefined generic function Unchecked_Conversion is not allowed.

5/2 {*AI95-00394-01*} No_Unchecked_Deallocation
Semantic dependence on the predefined generic procedure Unchecked_Deallocation is not allowed.

Wording Changes from Ada 95

5.a/3 {*AI95-00394-01*} {*AI05-0299-1*} This subclause is new. These restrictions are replaced by the more general No_Dependence (see 13.12.1).

J.14 Character and Wide_Character Conversion Functions

Static Semantics

1/2 {*AI95-00395-01*} The following declarations exist in the declaration of package Ada.Characters.Handling:

2/2
```
function Is_Character (Item : in Wide_Character) return Boolean
   renames Conversions.Is_Character;
function Is_String    (Item : in Wide_String)    return Boolean
   renames Conversions.Is_String;
```

3/2
```
function To_Character (Item       : in Wide_Character;
                       Substitute : in Character := ' ')
                       return Character
   renames Conversions.To_Character;
```

4/2
```
function To_String    (Item       : in Wide_String;
                        Substitute : in Character := ' ')
                        return String
   renames Conversions.To_String;
```

5/2
```
function To_Wide_Character (Item : in Character) return Wide_Character
   renames Conversions.To_Wide_Character;
```

6/2
```
function To_Wide_String    (Item : in String)    return Wide_String
   renames Conversions.To_Wide_String;
```

Wording Changes from Ada 95

6.a/3 {*AI95-00394-01*} {*AI05-0299-1*} This subclause is new. These subprograms were moved to Characters.Conversions (see A.3.4).

J.15 Aspect-related Pragmas

1/3 {*AI05-0229-1*} Pragmas can be used as an alternative to aspect_specifications to specify certain aspects.

Name Resolution Rules

{*AI12-0417-1*} Certain pragmas are defined to be *program unit pragmas*. A name given as the argument of a program unit pragma shall resolve to denote the declarations or renamings of one or more program units that occur immediately within the declarative region or compilation in which the pragma immediately occurs, or it shall resolve to denote the declaration of the immediately enclosing program unit (if any); the pragma applies to the denoted program unit(s). If there are no names given as arguments, the pragma applies to the immediately enclosing program unit. 2/5

Ramification: The fact that this is a Name Resolution Rule means that the pragma will not apply to declarations from outer declarative regions. 2.a/5

Legality Rules

{*AI12-0417-1*} A program unit pragma shall appear in one of these places: 3/5

- {*AI12-0417-1*} At the place of a compilation_unit, in which case the pragma shall immediately follow in the same compilation (except for other pragmas) a library_unit_declaration that is a subprogram_declaration, generic_subprogram_declaration, or generic_instantiation, and the pragma shall have an argument that is a name denoting that declaration. 4/5

  Ramification: The name has to denote the immediately preceding library_unit_declaration. 4.a/5

- {*8652/0033*} {*AI95-00136-01*} {*AI12-0417-1*} Immediately within the visible part of a program unit and before any nested declaration (but not within a generic formal part), in which case the argument, if any, shall be a direct_name that denotes the immediately enclosing program unit declaration. 5/5

  Ramification: The argument is optional in this case. 5.a/5

- {*AI12-0417-1*} At the place of a declaration other than the first, of a declarative_part or program unit declaration, in which case the pragma shall have an argument, which shall be a direct_name that denotes one or more of the following (and nothing else): a subprogram_declaration, a generic_subprogram_declaration, or a generic_instantiation, of the same declarative_part or program unit declaration. 6/5

  Ramification: If you want to denote a subprogram_body that is not a completion, or a package_declaration, for example, you have to put the pragma inside. 6.a/5

{*AI05-0132-1*} {*AI12-0417-1*} Certain program unit pragmas are defined to be *library unit pragmas*. If a library unit pragma applies to a program unit, the program unit shall be a library unit. 7/5

Ramification: This, together with the rules for program unit pragmas above, implies that if a library unit pragma applies to a subprogram_declaration (and similar things), it has to appear immediately after the compilation_unit, whereas if the pragma applies to a package_declaration, a subprogram_body that is not a completion (and similar things), it has to appear inside, as the first declarative_item. 7.a/5

Static Semantics

{*8652/0034*} {*AI95-00041-01*} {*AI12-0417-1*} A library unit pragma that applies to a generic unit does not apply to its instances, unless a specific rule for the pragma specifies the contrary. 8/5

Implementation Advice

{*8652/0034*} {*AI95-00041-01*} {*AI12-0417-1*} When applied to a generic unit, a program unit pragma that is not a library unit pragma should apply to each instance of the generic unit for which there is not an overriding pragma applied directly to the instance. 9/5

Implementation Advice: When applied to a generic unit, a program unit pragma that is not a library unit pragma should apply to each instance of the generic unit for which there is not an overriding pragma applied directly to the instance. 9.a/5

Wording Changes from Ada 2005

9.b/3 {*AI05-0229-1*} {*AI05-0299-1*} This subclause is new. Many existing pragmas have been converted into aspects; the pragmas have moved here.

9.c/5 {*AI05-0132-1*} {*AI12-0417-1*} **Correction:** A library unit pragma must apply directly to a library unit, even if no name is given in the pragma.

Wording Changes from Ada 2012

9.d/5 {*AI12-0417-1*} The terms “program unit pragma” and “library unit pragma” were moved here as all of the pragmas that use these terms are now in this annex.

J.15.1 Pragma Inline

Syntax

1/3 {*AI05-0229-1*} The form of a pragma Inline, which is a program unit pragma (see 10.1.5), is as follows:

2/3 **pragma** Inline (name{, name});

Legality Rules

3/3 {*AI05-0229-1*} The pragma shall apply to one or more callable entities or generic subprograms.

Static Semantics

4/3 {*AI05-0229-1*} Pragma Inline specifies that the Inline aspect (see 6.3.2) for each entity denoted by each name given in the pragma has the value True.

4.a/3 **Ramification:** Note that inline expansion is desired no matter what name is used in the call. This allows one to request inlining for only one of several overloaded subprograms as follows:

4.b/3
```
package IO is
   procedure Put(X : in Integer);
   procedure Put(X : in String);
   procedure Put(X : in Character);
private
   procedure Character_Put(X : in Character) renames Put;
   pragma Inline(Character_Put);
end IO;
```

4.c/3
```
with IO; use IO;
procedure Main is
   I : Integer;
   C : Character;
begin
   ...
   Put(C); -- Inline expansion is desired.
   Put(I); -- Inline expansion is NOT desired.
end Main;
```

Implementation Permissions

5/3 {*AI05-0229-1*} An implementation may allow a pragma Inline that has an argument which is a direct_name denoting a subprogram_body of the same declarative_part.

5.a/3 **Reason:** This is allowed for Ada 83 compatibility. This is only a permission as this usage was considered obsolescent even for Ada 95.

5.b/3 **Discussion:** We only need to allow this in declarative_parts, because a body is only allowed in another body, and these all have declarative_parts.

6/5 NOTE {*AI05-0229-1*} {*AI12-0440-1*} The name in a pragma Inline can denote more than one entity in the case of overloading. Such a pragma applies to all of the denoted entities.

Incompatibilities With Ada 83

{*AI95-00309-01*} {*AI05-0229-1*} A pragma Inline cannot refer to a subprogram_body outside of that body. The pragma can be given inside of the subprogram body. Ada 2005 adds an Implementation Permission to allow this usage for compatibility (and Ada 95 implementations also can use this permission), but implementations do not have to allow such pragmas. 6.a/3

Extensions to Ada 83

{*AI05-0229-1*} A pragma Inline is allowed inside a subprogram_body if there is no corresponding subprogram_declaration. This is for uniformity with other program unit pragmas. 6.b/3

Extensions to Ada 95

{*AI95-00309-01*} {*AI05-0229-1*} **Amendment Correction:** Implementations are allowed to let Pragma Inline apply to a subprogram_body. 6.c/3

Wording Changes from Ada 2005

{*AI05-0229-1*} {*AI05-0299-1*} This subclause is new. Pragma Inline was moved here from 6.3.2; aspect Inline lives there now. 6.d/3

J.15.2 Pragma No_Return

Syntax

{*AI05-0229-1*} The form of a pragma No_Return, which is a representation pragma (see 13.1), is as follows: 1/3

{*AI12-0269-1*} **pragma** No_Return (*subprogram*_local_name{, *subprogram*_local_name}); 2/5

Legality Rules

{*AI05-0229-1*} {*AI12-0269-1*} Each *subprogram*_local_name shall denote one or more subprograms or generic subprograms. [The *subprogram*_local_name shall not denote a null procedure nor an instance of a generic unit.] 3/5

Static Semantics

{*AI05-0229-1*} {*AI12-0269-1*} Pragma No_Return specifies that the No_Return aspect (see 6.5.1) for each subprogram denoted by each local_name given in the pragma has the value True. 4/5

Wording Changes from Ada 2005

{*AI05-0229-1*} {*AI05-0299-1*} This subclause is new. Pragma No_Return was moved here from 6.5.1; aspect No_Return lives there now. 4.a/3

Wording Changes from Ada 2012

{*AI12-0269-1*} This subclause is new. Pragma No_Return was moved here from 6.5.1; aspect No_Return lives there now. 4.b/5

J.15.3 Pragma Pack

Syntax

{*AI05-0229-1*} The form of a pragma Pack, which is a representation pragma (see 13.1), is as follows: 1/3

pragma Pack (*first_subtype*_local_name); 2/3

Legality Rules

3/3 {*AI05-0229-1*} The *first_subtype*_local_name of a pragma Pack shall denote a composite subtype.

Static Semantics

4/3 {*AI05-0229-1*} Pragma Pack specifies that the Pack aspect (see 13.2) for the type denoted by *first_subtype*_local_name has the value True.

Wording Changes from Ada 2005

4.a/3 {*AI05-0229-1*} {*AI05-0299-1*} This subclause is new. Pragma Pack was moved here from 13.2; aspect Pack lives there now.

J.15.4 Pragma Storage_Size

Syntax

1/3 {*AI05-0229-1*} The form of a pragma Storage_Size is as follows:

2/3 **pragma** Storage_Size (expression);

3/3 {*AI05-0229-1*} A pragma Storage_Size is allowed only immediately within a task_definition.

Name Resolution Rules

4/3 {*AI05-0229-1*} The expression of a pragma Storage_Size is expected to be of any integer type.

Static Semantics

5/3 {*AI05-0229-1*} The pragma Storage_Size sets the Storage_Size aspect (see 13.3) of the type defined by the immediately enclosing task_definition to the value of the expression of the pragma.

Wording Changes from Ada 2005

5.a/3 {*AI05-0229-1*} {*AI05-0299-1*} This subclause is new. Pragma Storage_Size was moved here from 13.3; aspect Storage_Size lives there now.

J.15.5 Interfacing Pragmas

Syntax

1/3 {*AI05-0229-1*} An *interfacing pragma* is a representation pragma that is one of the pragmas Import, Export, or Convention. Their forms are as follows:

2/3 **pragma** Import(
[Convention =>] *convention*_identifier, [Entity =>] local_name
[, [External_Name =>] *external_name_string*_expression]
[, [Link_Name =>] *link_name_string*_expression]);

3/3 **pragma** Export(
[Convention =>] *convention*_identifier, [Entity =>] local_name
[, [External_Name =>] *external_name_string*_expression]
[, [Link_Name =>] *link_name_string*_expression]);

4/3 **pragma** Convention([Convention =>] *convention*_identifier,[Entity =>] local_name);

5/3 {*AI05-0229-1*} For pragmas Import and Export, the argument for Link_Name shall not be given without the *pragma_argument*_identifier unless the argument for External_Name is given.

Name Resolution Rules

{*AI05-0229-1*} The expected type for an *external_name_string*_expression and a *link_name_string*_expression in an interfacing pragma is String. 6/3

Legality Rules

{*AI05-0229-1*} The *convention*_identifier of an interfacing pragma shall be the name of a convention (see B.1). 7/3

{*AI05-0229-1*} A pragma Import shall be the completion of a declaration. Notwithstanding any rule to the contrary, a pragma Import may serve as the completion of any kind of (explicit) declaration if supported by an implementation for that kind of declaration. If a completion is a pragma Import, then it shall appear in the same declarative_part, package_specification, task_definition, or protected_definition as the declaration. For a library unit, it shall appear in the same compilation, before any subsequent compilation_units other than pragmas. If the local_name denotes more than one entity, then the pragma Import is the completion of all of them. 8/3

{*AI05-0229-1*} The *external_name_string*_expression and *link_name_string*_expression of a pragma Import or Export shall be static. 9/3

{*AI05-0229-1*} The local_name of each of these pragmas shall denote a declaration that may have the similarly named aspect specified. 10/3

Static Semantics

{*AI05-0229-1*} An interfacing pragma specifies various aspects of the entity denoted by the local_name as follows: 11/3

- The Convention aspect (see B.1) is *convention*_identifier. 12/3
- A pragma Import specifies that the Import aspect (see B.1) is True. 13/3
- A pragma Export specifies that the Export aspect (see B.1) is True. 14/3
- For both pragma Import and Export, if an external name is given in the pragma, the External_Name aspect (see B.1) is specified to be *external_name_string*_expression. If a link name is given in the pragma, the Link_Name aspect (see B.1) is specified to be the *link_name_string*_expression. 15/3

Wording Changes from Ada 2005

{*AI05-0229-1*} {*AI05-0299-1*} This subclause is new. Pragmas Import, Export, and Convention were moved here from B.1; aspects Import, Export, Convention, Link_Name, and External_Name live there now. 15.a/3

J.15.6 Pragma Unchecked_Union

Syntax

{*AI05-0229-1*} The form of a pragma Unchecked_Union, which is a representation pragma (see 13.1), is as follows: 1/3

pragma Unchecked_Union (*first_subtype*_local_name); 2/3

Legality Rules

{*AI05-0229-1*} The *first_subtype*_local_name of a pragma Unchecked_Union shall denote an unconstrained discriminated record subtype having a variant_part. 3/3

Static Semantics

4/3 {*AI05-0229-1*} A pragma Unchecked_Union specifies that the Unchecked_Union aspect (see B.3.3) for the type denoted by *first_subtype*_local_name has the value True.

Wording Changes from Ada 2005

4.a/3 {*AI05-0229-1*} {*AI05-0299-1*} This subclause is new. Pragma Unchecked_Union was moved here from B.3.3; aspect Unchecked_Union lives there now.

J.15.7 Pragmas Interrupt_Handler and Attach_Handler

Syntax

1/3 {*AI05-0229-1*} The form of a pragma Interrupt_Handler is as follows:

2/3 **pragma** Interrupt_Handler (*handler*_name);

3/3 {*AI05-0229-1*} The form of a pragma Attach_Handler is as follows:

4/3 **pragma** Attach_Handler (*handler*_name, expression);

Name Resolution Rules

5/3 {*AI05-0229-1*} For the Interrupt_Handler and Attach_Handler pragmas, the *handler*_name shall resolve to denote a protected procedure with a parameterless profile.

6/3 {*AI05-0229-1*} For the Attach_Handler pragma, the expected type for the expression is Interrupts.Interrupt_Id (see C.3.2).

Legality Rules

7/3 {*AI05-0033-1*} {*AI05-0229-1*} The Attach_Handler and Interrupt_Handler pragmas are only allowed immediately within the protected_definition where the corresponding subprogram is declared. The corresponding protected_type_declaration or single_protected_declaration shall be a library-level declaration, and shall not be declared within a generic body. In addition to the places where Legality Rules normally apply (see 12.3), these rules also apply in the private part of an instance of a generic unit.

7.a/3 **Discussion:** In the case of a protected_type_declaration, an object_declaration of an object of that type need not be at library level.

7.b/3 {*AI05-0033-1*} We cannot allow these pragmas in a generic body, because legality rules are not checked for instance bodies, and these should not be allowed if the instance is not at the library level. The protected types can be declared in the private part if this is desired. Note that while the 'Access to use the handler would provide the check in the case of Interrupt_Handler, there is no other check for Attach_Handler. Since these pragmas are so similar, we want the rules to be the same.

Static Semantics

8/3 {*AI05-0229-1*} For an implementation that supports Annex C, a pragma Interrupt_Handler specifies the Interrupt_Handler aspect (see C.3.1) for the protected procedure *handler*_name to have the value True. For an implementation that supports Annex C, a pragma Attach_Handler specifies the Attach_Handler aspect (see C.3.1) for the protected procedure *handler*_name to have the value of the given expression[as evaluated at object creation time].

Incompatibilities With Ada 2005

8.a/3 {*AI05-0033-1*} **Correction:** Added missing generic contract wording for the pragma Attach_Handler and Interrupt_Handler. This means that nested instances with these pragmas in the private part are now illegal. This is not likely to occur in practice.

Wording Changes from Ada 2005

{*AI05-0229-1*} {*AI05-0299-1*} This subclause is new. Pragmas Interrupt_Handler and Attach_Handler were moved here from C.3.1; aspects Interrupt_Handler and Attach_Handler live there now. 8.b/3

J.15.8 Shared Variable Pragmas

Syntax

{*AI05-0229-1*} {*AI12-0425-1*} The following pragmas are defined with the given forms: 1/5

pragma Atomic (local_name); 2/3

pragma Volatile (local_name); 3/3

{*AI05-0009-1*} **pragma** Independent (*component*_local_name); 4/3

pragma Atomic_Components (*array*_local_name); 5/3

pragma Volatile_Components (*array*_local_name); 6/3

{*AI05-0009-1*} **pragma** Independent_Components (local_name); 7/3

Discussion: {*AI05-0009-1*} {*AI05-0229-1*} Pragmas Independent and Independent_Components are born obsolescent; they are defined to provide consistency with the existing shared variable pragmas. As with all obsolescent features, these pragmas are not optional; all Ada implementations need to implement them. Also note that these pragmas were defined as a **Correction**; as such, they are expected to be implemented as part of Ada 2005 implementations (and they would not be obsolescent there). 7.a/3

Name Resolution Rules

{*AI05-0009-1*} {*AI05-0229-1*} The local_name in an Atomic or Volatile pragma shall resolve to denote either an object_declaration, a noninherited component_declaration, or a full_type_declaration. The *component*_local_name in an Independent pragma shall resolve to denote a noninherited component_declaration. The *array*_local_name in an Atomic_Components or Volatile_Components pragma shall resolve to denote the declaration of an array type or an array object of an anonymous type. The local_name in an Independent_Components pragma shall resolve to denote the declaration of an array or record type or an array object of an anonymous type. 8/3

Static Semantics

{*AI05-0229-1*} These pragmas are representation pragmas (see 13.1). Each of these pragmas specifies that the similarly named aspect (see C.6) of the type, object, or component denoted by its argument is True. 9/3

Legality Rules

{*AI05-0229-1*} The local_name of each of these pragmas shall denote a declaration that may have the similarly named aspect specified. 10/3

Wording Changes from Ada 2005

{*AI05-0229-1*} {*AI05-0299-1*} This subclause is new. These pragmas were moved here from C.6; various aspects live there now. 10.a/3

J.15.9 Pragma CPU

Discussion: {*AI05-0229-1*} This pragma is born obsolescent; it is defined to provide consistency with existing real-time pragmas. As with all obsolescent features, this pragma is not optional; all Ada implementations need to implement it. 0.a/3

Syntax

1/3 {*AI05-0229-1*} The form of a pragma CPU is as follows:

2/3 **pragma** CPU (expression);

Name Resolution Rules

3/3 {*AI05-0229-1*} The expected type for the expression of a pragma CPU is System.Multiprocessors.CPU_Range.

Legality Rules

4/5 {*AI05-0229-1*} {*AI12-0281-1*} A CPU pragma is allowed only immediately within a task_definition, protected_definition, or the declarative_part of a subprogram_body.

5/3 {*AI05-0229-1*} For a CPU pragma that appears in the declarative_part of a subprogram_body, the expression shall be static.

Static Semantics

6/5 {*AI05-0229-1*} {*AI12-0281-1*} For an implementation that supports Annex D, a pragma CPU specifies the value of the CPU aspect (see D.16). If the pragma appears in a task_definition, the expression is associated with the aspect for the task type or single_task_declaration that contains the pragma. If the pragma appears in a protected_definition, the expression is associated with the aspect for the protected type or single_protected_declaration that contains the pragma. Otherwise, the expression is associated with the aspect for the subprogram that contains the pragma.

Extensions to Ada 2005

6.a/3 {*AI05-0009-1*} Pragma CPU is new.

J.15.10 Pragma Dispatching_Domain

0.a/3 **Discussion:** {*AI05-0167-1*} This pragma is born obsolescent; it is defined to provide consistency with existing real-time pragmas. As with all obsolescent features, this pragma is not optional; all Ada implementations need to implement it.

Syntax

1/3 {*AI05-0167-1*} The form of a pragma Dispatching_Domain is as follows:

2/3 **pragma** Dispatching_Domain (expression);

Name Resolution Rules

3/3 {*AI05-0167-1*} The expected type for the expression is System.Multiprocessors.Dispatching_Domains.-Dispatching_Domain.

Legality Rules

4/3 {*AI05-0167-1*} A Dispatching_Domain pragma is allowed only immediately within a task_definition.

Static Semantics

5/3 {*AI05-0167-1*} For an implementation that supports Annex D, a pragma Dispatching_Domain specifies the value of the Dispatching_Domain aspect (see D.16.1). The expression is associated with the aspect for the task type or single_task_declaration that contains the pragma.

Extensions to Ada 2005

{*AI05-0009-1*} Pragma Dispatching_Domain is new. 5.a/3

J.15.11 Pragmas Priority and Interrupt_Priority

Syntax

{*AI05-0229-1*} The form of a pragma Priority is as follows: 1/3

pragma Priority (expression); 2/3

{*AI05-0229-1*} The form of a pragma Interrupt_Priority is as follows: 3/3

pragma Interrupt_Priority [(expression);] 4/3

Name Resolution Rules

{*AI05-0229-1*} The expected type for the expression in a Priority or Interrupt_Priority pragma is Integer. 5/3

Legality Rules

{*AI05-0229-1*} A Priority pragma is allowed only immediately within a task_definition, a protected_definition, or the declarative_part of a subprogram_body. An Interrupt_Priority pragma is allowed only immediately within a task_definition or a protected_definition. 6/3

{*AI05-0229-1*} For a Priority pragma that appears in the declarative_part of a subprogram_body, the expression shall be static, and its value shall be in the range of System.Priority. 7/3

Static Semantics

{*AI05-0229-1*} For an implementation that supports Annex D, a pragma Priority specifies the value of the Priority aspect (see D.1) and a pragma Interrupt_Priority specifies the value of the Interrupt_Priority aspect as follows: 8/3

- If the pragma appears in a task_definition, the expression is associated with the aspect for the task type or single_task_declaration that contains the pragma; 9/3
- If the pragma appears in a protected_definition, the expression is associated with the aspect for the protected type or single_protected_declaration that contains the pragma; 10/3
- If the pragma appears in the declarative_part of a subprogram_body, the expression is associated with the aspect for the subprogram that contains the pragma. 11/3

{*AI05-0229-1*} If there is no expression in an Interrupt_Priority pragma, the Interrupt_Priority aspect has the value Interrupt_Priority'Last. 12/3

Wording Changes from Ada 2005

{*AI05-0229-1*} {*AI05-0299-1*} This subclause is new. Pragmas Interrupt_Priority and Priority were moved here from D.1; aspects Interrupt_Priority and Priority live there now. 12.a/3

J.15.12 Pragma Relative_Deadline

Syntax

{*AI05-0229-1*} The form of a pragma Relative_Deadline is as follows: 1/3

pragma Relative_Deadline (*relative_deadline*_expression); 2/3

Name Resolution Rules

3/3 {*AI05-0229-1*} The expected type for a *relative_deadline*_expression is Real_Time.Time_Span.

Legality Rules

4/3 {*AI05-0229-1*} A Relative_Deadline pragma is allowed only immediately within a task_definition or the declarative_part of a subprogram_body.

Static Semantics

5/3 {*AI05-0229-1*} For an implementation that supports Annex D, a pragma Relative_Deadline specifies the value of the Relative_Deadline aspect (see D.2.6). If the pragma appears in a task_definition, the expression is associated with the aspect for the task type or single_task_declaration that contains the pragma; otherwise, the expression is associated with the aspect for the subprogram that contains the pragma.

Wording Changes from Ada 2005

5.a/3 {*AI05-0229-1*} {*AI05-0299-1*} This subclause is new. Pragma Relative_Deadline was moved here from D.2.6; aspect Relative_Deadline lives there now.

J.15.13 Pragma Asynchronous

Syntax

1/3 {*AI05-0229-1*} The form of a pragma Asynchronous, which is a representation pragma (see 13.1), is as follows:

2/3 **pragma** Asynchronous (local_name);

Static Semantics

3/3 {*AI05-0229-1*} For an implementation that supports Annex E, a pragma Asynchronous specifies that the Asynchronous aspect (see E.4.1) for the procedure or type denoted by local_name has the value True.

Legality Rules

4/3 {*AI05-0229-1*} The local_name of a pragma Asynchronous shall denote a declaration that may have aspect Asynchronous specified.

Wording Changes from Ada 2005

4.a/3 {*AI05-0229-1*} {*AI05-0299-1*} This subclause is new. Pragma Asynchronous was moved here from E.4.1; aspect Asynchronous lives there now.

J.15.14 Elaboration Control Pragmas

1/5 {*AI12-0409-1*} This subclause defines pragmas that specify aspects that help control the elaboration order of library_items.

1.a/5 **Discussion:** {*AI12-0417-1*} Pragmas that do not have associated aspects still appear in 10.2.1. These pragmas are presented in the order that the aspects are defined in 10.2.1.

Syntax

2/5 {*AI12-0417-1*} The following pragmas are defined with the given forms:

3/5 **pragma** Preelaborate[(*library_unit*_name)];

4/5 **pragma** Preelaborable_Initialization(direct_name);

pragma Pure[(*library_unit*_name)]; 5/5

pragma Elaborate_Body[(*library_unit*_name)]; 6/5

{*AI12-0417-1*} Pragmas Preelaborate, Pure, and Elaborate_Body are library unit pragmas. 7/5

Ramification: Pragma Preelaborable_Initialization is **not** a library unit pragma. 7.a/5

Static Semantics

{*AI12-0417-1*} A pragma Preelaborate specifies that a library unit is preelaborated, namely that the Preelaborate aspect (see 10.2.1) of the library unit is True. 8/5

{*AI12-0417-1*} A pragma Pure specifies that a library unit is declared pure, namely that the Pure aspect (see 10.2.1) of the library unit is True. 9/5

{*AI12-0417-1*} A pragma Elaborate_Body specifies that [a library unit requires a completion, namely that] the Elaborate_Body aspect (see 10.2.1) of the library unit is True. 10/5

Legality Rules

{*AI12-0409-1*} A pragma Preelaborable_Initialization specifies that the Preelaborable_Initialization aspect (see 10.2.1) for a composite type is True. This pragma shall appear in the visible part of a package or generic package. 11/5

{*AI12-0409-1*} If the pragma appears in the first declaration list of a package_specification, then the direct_name shall denote the first subtype of a composite type, and the type shall be declared immediately within the same package as the pragma. The composite type shall be one for which the Preelaborable_Initialization aspect can be directly specified as True. In addition to the places where Legality Rules normally apply (see 12.3), these rules also apply in the private part of an instance of a generic unit. 12/5

{*AI12-0409-1*} If the pragma appears in a generic_formal_part, then the direct_name shall denote a type declared in the same generic_formal_part as the pragma, and be one for which the Preelaborable_Initialization aspect can be directly specified as True. 13/5

NOTE {*AI12-0409-1*} Pragmas Elaborate and Elaborate_All, which do not have associated aspects, are found in 10.2.1. 14/5

Wording Changes from Ada 2012

{*AI12-0409-1*} This subclause is new. These pragmas were moved here from 10.2.1. 14.a/5

J.15.15 Distribution Pragmas

{*AI12-0417-1*} This subclause defines pragmas that specify properties of units for distributed systems. 1/5

Discussion: {*AI12-0417-1*} These pragmas are presented in the order that the aspects are defined in Annex E. 1.a/5

Syntax

{*AI12-0417-1*} The following pragmas are defined with the given forms: 2/5

pragma Shared_Passive[(*library_unit*_name)]; 3/5

pragma Remote_Types[(*library_unit*_name)]; 4/5

pragma Remote_Call_Interface[(*library_unit*_name)]; 5/5

pragma All_Calls_Remote[(*library_unit*_name)]; 6/5

{*AI12-0417-1*} Each of these pragmas is a library unit pragma. 7/5

Static Semantics

8/5 {*AI12-0417-1*} A *categorization pragma* is a pragma that specifies a corresponding categorization aspect.

9/5 {*AI12-0417-1*} The pragmas Shared_Passive, Remote_Types, and Remote_Call_Interface are categorization pragmas. In addition, the pragma Pure (see J.15.14) is considered a categorization pragma.

10/5 {*AI12-0417-1*} A pragma Shared_Passive specifies that a library unit is a shared passive library unit, namely that the Shared_Passive aspect (see E.2.1) of the library unit is True.

11/5 {*AI12-0417-1*} A pragma Remote_Types specifies that a library unit is a remote types library unit, namely that the Remote_Types aspect (see E.2.2) of the library unit is True.

12 A pragma Remote_Call_Interface specifies that a library unit is a remote call interface, namely that the Remote_Call_Interface aspect (see E.2.3) of the library unit is True.

13 A pragma All_Calls_Remote specifies that the All_Calls_Remote aspect (see E.2.3) of the library unit is True.

Wording Changes from Ada 2012

13.a/5 {*AI12-0417-1*} This subclause is new. These pragmas were moved here from Annex E, “Distributed Systems”.

Annex K
(informative)
Language-Defined Aspects and Attributes

{*AI05-0229-1*} This annex summarizes the definitions given elsewhere of the language-defined aspects and attributes. Some aspects have corresponding attributes, as noted. 1/3

K.1 Language-Defined Aspects

{*AI05-0229-1*} {*AI05-0299-1*} This subclause summarizes the definitions given elsewhere of the language-defined aspects. Aspects are properties of entities that can be specified by the Ada program; unless otherwise specified below, aspects can be specified using an aspect_specification. 1/3

Address Machine address of an entity. See 13.3. 2/3

Aggregate Mechanism to define user-defined aggregates. See 4.3.5. 2.1/5

Alignment (object) 3/3
Alignment of an object. See 13.3.

Alignment (subtype) 4/3
Alignment of a subtype. See 13.3.

All_Calls_Remote 5/4
All indirect or dispatching remote subprogram calls, and all direct remote subprogram calls, should use the Partition Communication Subsystem. See E.2.3.

Allows_Exit 5.1/5
An indication of whether a subprogram will operate correctly for arbitrary transfers of control. See 5.5.3.

Asynchronous 6/3
Remote procedure calls are asynchronous; the caller continues without waiting for the call to return. See E.4.1.

Atomic Declare that a type, object, or component is atomic. See C.6. 7/3

Atomic_Components 8/3
Declare that the components of an array type or object are atomic. See C.6.

Attach_Handler 9/3
Protected procedure is attached to an interrupt. See C.3.1.

Bit_Order Order of bit numbering in a record_representation_clause. See 13.5.3. 10/3

Coding Internal representation of enumeration literals. Specified by an enumeration_representation_clause, not by an aspect_specification. See 13.4. 11/3

Component_Size 12/3
Size in bits of a component of an array type. See 13.3.

Constant_Indexing 13/3
Defines function(s) to implement user-defined indexed_components. See 4.1.6.

Convention Calling convention or other convention used for interfacing to other languages. See B.1. 14/3

CPU Processor on which a given task, or calling task for a protected operation, should run. See D.16. 15/5

16/3 Default_Component_Value
Default value for the components of an array-of-scalar subtype. See 3.6.

16.1/5 Default_Initial_Condition
A condition that will hold true after the default initialization of an object. See 7.3.3.

17/3 Default_Iterator
Default iterator to be used in **for** loops. See 5.5.1.

18/3 Default_Storage_Pool
Default storage pool for a generic instance. See 13.11.3.

19/3 Default_Value
Default value for a scalar subtype. See 3.5.

19.1/4 Discard_Names
Requests a reduction in storage for names associated with an entity. See C.5.

19.2/5 Dispatching
Generic formal parameters used in the implementation of an entity. See H.7.1.

20/3 Dispatching_Domain
Domain (group of processors) on which a given task should run. See D.16.1.

21/5 Dynamic_Predicate
Condition that will hold true for objects of a given subtype; the subtype is not static. See 3.2.4.

22/5 Elaborate_Body
A given package will have a body, and that body is elaborated immediately after the declaration. See 10.2.1.

22.1/4 Exclusive_Functions
Specifies mutual exclusion behavior of protected functions in a protected type. See 9.5.1.

23/3 Export Entity is exported to another language. See B.1.

24/3 External_Name
Name used to identify an imported or exported entity. See B.1.

25/3 External_Tag
Unique identifier for a tagged type in streams. See 13.3.

25.1/5 Full_Access_Only
Declare that a volatile type, object, or component is full access. See C.6.

25.2/5 Global Global object usage contract. See 6.1.2.

25.3/5 Global'Class
Global object usage contract inherited on derivation. See 6.1.2.

26/3 Implicit_Dereference
Mechanism for user-defined implicit .**all**. See 4.1.5.

27/3 Import Entity is imported from another language. See B.1.

28/3 Independent Declare that a type, object, or component is independently addressable. See C.6.

29/3 Independent_Components
Declare that the components of an array or record type, or an array object, are independently addressable. See C.6.

30/3 Inline For efficiency, Inline calls are requested for a subprogram. See 6.3.2.

Input — Function to read a value from a stream for a given type, including any bounds and discriminants. See 13.13.2. 31/3

Input'Class — 31.1/4
Function to read a value from a stream for a the class-wide type associated with a given type, including any bounds and discriminants. See 13.13.2.

Integer_Literal — 31.2/5
Defines a function to implement user-defined integer literals. See 4.2.1.

Interrupt_Handler — 32/3
Protected procedure may be attached to interrupts. See C.3.1.

Interrupt_Priority — 33/3
Priority of a task object or type, or priority of a protected object or type; the priority is in the interrupt range. See D.1.

Iterator_Element — 34/3
Element type to be used for user-defined iterators. See 5.5.1.

Iterator_View — 34.1/5
An alternative type to used for container element iterators. See 5.5.1.

Layout (record) — 35/3
Layout of record components. Specified by a record_representation_clause, not by an aspect_specification. See 13.5.1.

Link_Name Linker symbol used to identify an imported or exported entity. See B.1. 36/3

Machine_Radix — 37/3
Radix (2 or 10) that is used to represent a decimal fixed point type. See F.1.

Max_Entry_Queue_Length — 37.1/5
The maximum entry queue length for a task type, protected type, or entry. See D.4.

No_Controlled_Parts — 37.2/5
A specification that a type and its descendants do not have controlled parts. See H.4.1.

No_Return A subprogram will not return normally. See 6.5.1. 38/5

Nonblocking — 38.1/5
Specifies that an associated subprogram does not block. See 9.5.

Output — Procedure to write a value to a stream for a given type, including any bounds and discriminants. See 13.13.2. 39/3

Output'Class — 39.1/4
Procedure to write a value to a stream for a the class-wide type associated with a given type, including any bounds and discriminants. See 13.13.2.

Pack — Minimize storage when laying out records and arrays. See 13.2. 40/3

Parallel_Calls — 40.1/5
Specifies whether a given subprogram is expected to be called in parallel. See 9.10.1.

Parallel_Iterator — 40.2/5
An indication of whether a subprogram may use multiple threads of control to invoke a loop body procedure. See 5.5.3.

Post — Postcondition; a condition that will hold true after a call. See 6.1.1. 41/5

Post'Class — Postcondition that applies to corresponding subprograms of descendant types. See 6.1.1. 42/5

Pre — Precondition; a condition that is expected to hold true before a call. See 6.1.1. 43/5

44/5 Pre'Class Precondition that applies to corresponding subprograms of descendant types. See 6.1.1.

44.1/4 Predicate_Failure
Action to be performed when a predicate check fails. See 3.2.4.

44.2/5 Preelaborable_Initialization
Declares that a type has preelaborable initialization. See 10.2.1.

45/3 Preelaborate
Code execution during elaboration is avoided for a given package. See 10.2.1.

46/3 Priority Priority of a task object or type, or priority of a protected object or type; the priority is not in the interrupt range. See D.1.

47/3 Pure Side effects are avoided in the subprograms of a given package. See 10.2.1.

47.1/5 Put_Image Procedure to define the image of a given type. See 4.10.

48/3 Read Procedure to read a value from a stream for a given type. See 13.13.2.

48.1/4 Read'Class Procedure to read a value from a stream for the class-wide type associated with a given type. See 13.13.2.

48.2/5 Real_Literal
Defines a function or functions to implement user-defined real literals. See 4.2.1.

49/3 Record layout
See Layout. See 13.5.1.

50/5 Relative_Deadline
Task or protected type parameter used in Earliest Deadline First Dispatching. See D.2.6.

51/3 Remote_Call_Interface
Subprograms in a given package may be used in remote procedure calls. See E.2.3.

52/3 Remote_Types
Types in a given package may be used in remote procedure calls. See E.2.2.

53/3 Shared_Passive
A given package is used to represent shared memory in a distributed system. See E.2.1.

54/3 Size (object)
Size in bits of an object. See 13.3.

55/3 Size (subtype)
Size in bits of a subtype. See 13.3.

56/3 Small Scale factor for a fixed point type. See 3.5.10.

56.1/5 Stable_Properties
A list of functions describing characteristics that usually are unchanged by primitive operations of the type or an individual primitive subprogram. See 7.3.4.

56.2/5 Stable_Properties'Class
A list of functions describing characteristics that usually are unchanged by primitive operations of a class of types or a primitive subprogram for such a class. See 7.3.4.

56.3/5 Static Specifies that an associated expression function can be used in static expressions. See 6.8.

57/5 Static_Predicate
Condition that will hold true for objects of a given subtype; the subtype may be static. See 3.2.4.

58/3 Storage_Pool
Pool of memory from which **new** will allocate for a given access type. See 13.11.

Storage_Size (access) 59/3
Sets memory size for allocations for an access type. See 13.11.

Storage_Size (task) 60/3
Size in storage elements reserved for a task type or single task object. See 13.3.

Stream_Size 61/3
Size in bits used to represent elementary objects in a stream. See 13.13.2.

String_Literal 61.1/5
Defines a function to implement user-defined string literals. See 4.2.1.

Synchronization 62/5
Defines whether a given primitive operation of a synchronized interface will be implemented by an entry or protected procedure. See 9.5.

Type_Invariant 63/3
A condition that will hold true for all objects of a type. See 7.3.2.

Type_Invariant'Class 64/3
A condition that will hold true for all objects in a class of types. See 7.3.2.

Unchecked_Union 65/3
Type is used to interface to a C union type. See B.3.3.

Use_Formal 65.1/5
Generic formal parameters used in the implementation of an entity. See H.7.1.

Variable_Indexing 66/3
Defines function(s) to implement user-defined indexed_components. See 4.1.6.

Volatile Declare that a type, object, or component is volatile. See C.6. 67/3

Volatile_Components 68/3
Declare that the components of an array type or object are volatile. See C.6.

Write Procedure to write a value to a stream for a given type. See 13.13.2. 69/3

Write'Class 69.1/4
Procedure to write a value to a stream for a the class-wide type associated with a given type. See 13.13.2.

Yield Ensures that a callable entity includes a task dispatching point. See D.2.1. 69.2/5

K.2 Language-Defined Attributes

{*AI05-0229-1*} {*AI05-0299-1*} This subclause summarizes the definitions given elsewhere of the language-defined attributes. Attributes are properties of entities that can be queried by an Ada program. 1/3

P'Access For a prefix P that denotes a subprogram: 2

P'Access yields an access value that designates the subprogram denoted by P. The type of P'Access is an access-to-subprogram type (*S*), as determined by the expected type. See 3.10.2. 3

X'Access For a prefix X that denotes an aliased view of an object: 4

X'Access yields an access value that designates the object denoted by X. The type of X'Access is an access-to-object type, as determined by the expected type. The expected type shall be a general access type. See 3.10.2. 5

X'Address For a prefix X that denotes an object, program unit, or label: 6/1

7 Denotes the address of the first of the storage elements allocated to X. For a program unit or label, this value refers to the machine code associated with the corresponding body or statement. The value of this attribute is of type System.Address. See 13.3.

8 S'Adjacent For every subtype S of a floating point type *T*:

9 S'Adjacent denotes a function with the following specification:

10
```
function S'Adjacent (X, Towards : T)
  return T
```

11 If *Towards* = *X*, the function yields *X*; otherwise, it yields the machine number of the type *T* adjacent to *X* in the direction of *Towards*, if that machine number exists. If the result would be outside the base range of S, Constraint_Error is raised. When *T*'Signed_Zeros is True, a zero result has the sign of *X*. When *Towards* is zero, its sign has no bearing on the result. See A.5.3.

12 S'Aft For every fixed point subtype S:

13 S'Aft yields the number of decimal digits needed after the decimal point to accommodate the *delta* of the subtype S, unless the *delta* of the subtype S is greater than 0.1, in which case the attribute yields the value one. (S'Aft is the smallest positive integer N for which (10**N)*S'Delta is greater than or equal to one.) The value of this attribute is of the type *universal_integer*. See 3.5.10.

13.1/2 S'Alignment

For every subtype S:

13.2/2 The value of this attribute is of type *universal_integer*, and nonnegative.

13.3/2 For an object X of subtype S, if S'Alignment is not zero, then X'Alignment is a nonzero integral multiple of S'Alignment unless specified otherwise by a representation item. See 13.3.

14/1 X'Alignment

For a prefix X that denotes an object:

15/2 The value of this attribute is of type *universal_integer*, and nonnegative; zero means that the object is not necessarily aligned on a storage element boundary. If X'Alignment is not zero, then X is aligned on a storage unit boundary and X'Address is an integral multiple of X'Alignment (that is, the Address modulo the Alignment is zero).

16/2 See 13.3.

17 S'Base For every scalar subtype S:

18 S'Base denotes an unconstrained subtype of the type of S. This unconstrained subtype is called the *base subtype* of the type. See 3.5.

19 S'Bit_Order

For every specific record subtype S:

20 Denotes the bit ordering for the type of S. The value of this attribute is of type System.Bit_Order. See 13.5.3.

21/1 P'Body_Version

For a prefix P that statically denotes a program unit:

22 Yields a value of the predefined type String that identifies the version of the compilation unit that contains the body (but not any subunits) of the program unit. See E.3.

23 T'Callable For a prefix T that is of a task type (after any implicit dereference):

24 Yields the value True when the task denoted by T is *callable*, and False otherwise; See 9.9.

E'Caller For a prefix E that denotes an entry_declaration: 25

Yields a value of the type Task_Id that identifies the task whose call is now being serviced. Use of this attribute is allowed only inside an accept_statement, or entry_body after the entry_barrier, corresponding to the entry_declaration denoted by E. See C.7.1. 26/3

S'Ceiling For every subtype S of a floating point type *T*: 27

S'Ceiling denotes a function with the following specification: 28

```
function S'Ceiling (X : T)
  return T
```
29

The function yields the value $\lceil X \rceil$, that is, the smallest (most negative) integral value greater than or equal to *X*. When *X* is zero, the result has the sign of *X*; a zero result otherwise has a negative sign when S'Signed_Zeros is True. See A.5.3. 30/5

S'Class For every subtype S of a tagged type *T* (specific or class-wide): 31

S'Class denotes a subtype of the class-wide type (called *T*'Class in this document) for the class rooted at *T* (or if S already denotes a class-wide subtype, then S'Class is the same as S). 32

S'Class is unconstrained. However, if S is constrained, then the values of S'Class are only those that when converted to the type *T* belong to S. See 3.9. 33

S'Class For every subtype S of an untagged private type whose full view is tagged: 34

Denotes the class-wide subtype corresponding to the full view of S. This attribute is allowed only from the beginning of the private part in which the full view is declared, until the declaration of the full view. After the full view, the Class attribute of the full view can be used. See 7.3.1. 35

X'Component_Size 36/1

For a prefix X that denotes an array subtype or array object (after any implicit dereference):

Denotes the size in bits of components of the type of X. The value of this attribute is of type *universal_integer*. See 13.3. 37

S'Compose For every subtype S of a floating point type *T*: 38

S'Compose denotes a function with the following specification: 39

```
function S'Compose (Fraction : T;
                    Exponent : universal_integer)
  return T
```
40

Let *v* be the value $Fraction \cdot T\text{'Machine\_Radix}^{Exponent-k}$, where *k* is the normalized exponent of *Fraction*. If *v* is a machine number of the type *T*, or if $|v| \geq T\text{'Model\_Small}$, the function yields *v*; otherwise, it yields either one of the machine numbers of the type *T* adjacent to *v*. Constraint_Error is optionally raised if *v* is outside the base range of S. A zero result has the sign of *Fraction* when S'Signed_Zeros is True. See A.5.3. 41

A'Constrained 42

For a prefix A that is of a discriminated type (after any implicit dereference):

Yields the value True if A denotes a constant, a value, a tagged object, or a constrained variable, and False otherwise. The value of this attribute is of the predefined type Boolean. See 3.7.2. 43/5

S'Copy_Sign 44

For every subtype S of a floating point type *T*:

S'Copy_Sign denotes a function with the following specification: 45

```
function S'Copy_Sign (Value, Sign : T)
  return T
```
46

47 If the value of *Value* is nonzero, the function yields a result whose magnitude is that of *Value* and whose sign is that of *Sign*; otherwise, it yields the value zero. Constraint_Error is optionally raised if the result is outside the base range of S. A zero result has the sign of *Sign* when S'Signed_Zeros is True. See A.5.3.

48 E'Count For a prefix E that denotes an entry of a task or protected unit:

49 Yields the number of calls presently queued on the entry E of the current instance of the unit. The value of this attribute is of the type *universal_integer*. See 9.9.

50/1 S'Definite For a prefix S that denotes a formal indefinite subtype:

51/3 S'Definite yields True if the actual subtype corresponding to S is definite; otherwise, it yields False. The value of this attribute is of the predefined type Boolean. See 12.5.1.

52 S'Delta For every fixed point subtype S:

53 S'Delta denotes the *delta* of the fixed point subtype S. The value of this attribute is of the type *universal_real*. See 3.5.10.

54 S'Denorm For every subtype S of a floating point type *T*:

55 Yields the value True if every value expressible in the form

$$\pm\, mantissa \cdot T\text{'Machine\_Radix}^{T\text{'Machine\_Emin}}$$

where *mantissa* is a nonzero *T*'Machine_Mantissa-digit fraction in the number base *T*'Machine_Radix, the first digit of which is zero, is a machine number (see 3.5.7) of the type *T*; yields the value False otherwise. The value of this attribute is of the predefined type Boolean. See A.5.3.

56 S'Digits For every floating point subtype S:

57 S'Digits denotes the requested decimal precision for the subtype S. The value of this attribute is of the type *universal_integer*. See 3.5.8.

58 S'Digits For every decimal fixed point subtype S:

59 S'Digits denotes the *digits* of the decimal fixed point subtype S, which corresponds to the number of decimal digits that are representable in objects of the subtype. The value of this attribute is of the type *universal_integer*. See 3.5.10.

59.1/5 S'Enum_Rep

For every discrete subtype S:

59.2/5 S'Enum_Rep denotes a function with the following specification:

59.3/5
```
function S'Enum_Rep (Arg : S'Base) return universal_integer
```

59.4/5 This function returns the representation value of the value of Arg, as a value of type *universal_integer*. The *representation value* is the internal code specified in an enumeration representation clause, if any, for the type corresponding to the value of Arg, and otherwise is the position number of the value. See 13.4.

59.5/5 S'Enum_Val

For every discrete subtype S:

59.6/5 S'Enum_Val denotes a function with the following specification:

59.7/5
```
function S'Enum_Val (Arg : universal_integer) return S'Base
```

59.8/5 This function returns a value of the type of S whose representation value equals the value of Arg. For the evaluation of a call on S'Enum_Val, if there is no value in the base range of its type with the given representation value, Constraint_Error is raised. See 13.4.

60 S'Exponent For every subtype S of a floating point type *T*:

61 S'Exponent denotes a function with the following specification:

```
function S'Exponent (X : T)
  return universal_integer
```

62

The function yields the normalized exponent of *X*. See A.5.3. 63

S'External_Tag 64

For every subtype S of a tagged type *T* (specific or class-wide):

S'External_Tag denotes an external string representation for S'Tag; it is of the predefined type String. External_Tag may be specified for a specific tagged type via an attribute_definition_clause; the expression of such a clause shall be static. The default external tag representation is implementation defined. See 13.13.2. See 13.3. 65

A'First For a prefix A that is of an array type (after any implicit dereference), or denotes a constrained array subtype: 66/1

A'First denotes the lower bound of the first index range; its type is the corresponding index type. See 3.6.2. 67

S'First For every scalar subtype S: 68

S'First denotes the lower bound of the range of S. The value of this attribute is of the type of S. See 3.5. 69

A'First(N) 70/1

For a prefix A that is of an array type (after any implicit dereference), or denotes a constrained array subtype:

A'First(N) denotes the lower bound of the N-th index range; its type is the corresponding index type. See 3.6.2. 71

R.C'First_Bit 72

For a component C of a composite, non-array object R:

If the nondefault bit ordering applies to the composite type, and if a component_clause specifies the placement of C, denotes the value given for the first_bit of the component_clause; otherwise, denotes the offset, from the start of the first of the storage elements occupied by C, of the first bit occupied by C. This offset is measured in bits. The first bit of a storage element is numbered zero. The value of this attribute is of the type *universal_integer*. See 13.5.2. 73/2

S'First_Valid 73.1/4

For every static discrete subtype S for which there exists at least one value belonging to S that satisfies the predicates of S:

S'First_Valid denotes the smallest value that belongs to S and satisfies the predicates of S. The value of this attribute is of the type of S. See 3.5.5. 73.2/4

S'Floor For every subtype S of a floating point type *T*: 74

S'Floor denotes a function with the following specification: 75

```
function S'Floor (X : T)
  return T
```

76

The function yields the value $\lfloor X \rfloor$, that is, the largest (most positive) integral value less than or equal to *X*. When *X* is zero, the result has the sign of *X*; a zero result otherwise has a positive sign. See A.5.3. 77/5

S'Fore For every fixed point subtype S: 78

S'Fore yields the minimum number of characters needed before the decimal point for the decimal representation of any value of the subtype S, assuming that the representation does not include an exponent, but includes a one-character prefix that is either a minus sign or a 79

space. (This minimum number does not include superfluous zeros or underlines, and is at least 2.) The value of this attribute is of the type *universal_integer*. See 3.5.10.

80 S'Fraction For every subtype S of a floating point type *T*:

81 S'Fraction denotes a function with the following specification:

82
```
function S'Fraction (X : T)
  return T
```

83 The function yields the value $X \cdot T\text{'Machine\_Radix}^{-k}$, where *k* is the normalized exponent of *X*. A zero result, which can only occur when *X* is zero, has the sign of *X*. See A.5.3.

83.1/3 X'Has_Same_Storage
For a prefix X that denotes an object:

83.2/4 X'Has_Same_Storage denotes a function with the following specification:

83.3/3
```
function X'Has_Same_Storage (Arg : any_type)
  return Boolean
```

83.4/4 The actual parameter shall be a name that denotes an object. The object denoted by the actual parameter can be of any type. This function evaluates the names of the objects involved. It returns True if the representation of the object denoted by the actual parameter occupies exactly the same bits as the representation of the object denoted by X and the objects occupy at least one bit; otherwise, it returns False. See 13.3.

84/1 E'Identity For a prefix E that denotes an exception:

85 E'Identity returns the unique identity of the exception. The type of this attribute is Exception_Id. See 11.4.1.

86 T'Identity For a prefix T that is of a task type (after any implicit dereference):

87 Yields a value of the type Task_Id that identifies the task denoted by T. See C.7.1.

88 S'Image For every subtype S of a type T:

89/5 S'Image denotes a function with the following specification:

90/5
```
function S'Image(Arg : S'Base)
  return String
```

91/5 S'Image calls S'Put_Image passing *Arg* (which will typically store a sequence of character values in a text buffer) and then returns the result of retrieving the contents of that buffer with function Get. See 4.10.

91.1/5 X'Image For a prefix X of a type T other than *universal_real* or *universal_fixed*:

91.2/5 X'Image denotes the result of calling function S'Image with *Arg* being X, where S is the nominal subtype of X. See 4.10.

91.3/5 E'Index For a prefix E that denotes an entry declaration of an entry family:

91.4/5 Within a precondition or postcondition expression for entry family E, denotes the value of the entry index for the call of E. The nominal subtype of this attribute is the entry index subtype. See 6.1.1.

92 S'Class'Input
For every subtype S'Class of a class-wide type *T*'Class:

93 S'Class'Input denotes a function with the following specification:

94/2
```
function S'Class'Input(
   Stream : not null access Ada.Streams.Root_Stream_Type'Class)
   return T'Class
```

95/5 First reads the external tag from *Stream* and determines the corresponding internal tag (by calling Tags.Descendant_Tag(String'Input(*Stream*), S'Tag) which can raise Tag_Error — see

3.9) and then dispatches to the subprogram denoted by the Input attribute of the specific type identified by the internal tag; returns that result. If the specific type identified by the internal tag is abstract, Constraint_Error is raised. See 13.13.2.

S'Input For every subtype S of a specific type *T*: 96

S'Input denotes a function with the following specification: 97

```
function S'Input(
   Stream : not null access Ada.Streams.Root_Stream_Type'Class)
   return T
```
98/2

S'Input reads and returns one value from *Stream*, using any bounds or discriminants written by a corresponding S'Output to determine how much to read. See 13.13.2. 99

A'Last For a prefix A that is of an array type (after any implicit dereference), or denotes a constrained array subtype: 100/1

A'Last denotes the upper bound of the first index range; its type is the corresponding index type. See 3.6.2. 101

S'Last For every scalar subtype S: 102

S'Last denotes the upper bound of the range of S. The value of this attribute is of the type of S. See 3.5. 103

A'Last(N) For a prefix A that is of an array type (after any implicit dereference), or denotes a constrained array subtype: 104/1

A'Last(N) denotes the upper bound of the N-th index range; its type is the corresponding index type. See 3.6.2. 105

R.C'Last_Bit 106

For a component C of a composite, non-array object R:

If the nondefault bit ordering applies to the composite type, and if a component_clause specifies the placement of C, denotes the value given for the last_bit of the component_clause; otherwise, denotes the offset, from the start of the first of the storage elements occupied by C, of the last bit occupied by C. This offset is measured in bits. The value of this attribute is of the type *universal_integer*. See 13.5.2. 107/2

S'Last_Valid 107.1/4

For every static discrete subtype S for which there exists at least one value belonging to S that satisfies the predicates of S:

S'Last_Valid denotes the largest value that belongs to S and satisfies the predicates of S. The value of this attribute is of the type of S. See 3.5.5. 107.2/4

S'Leading_Part 108

For every subtype S of a floating point type *T*:

S'Leading_Part denotes a function with the following specification: 109

```
function S'Leading_Part (X : T;
                         Radix_Digits : universal_integer)
  return T
```
110

Let *v* be the value *T*'Machine_Radix$^{k-Radix_Digits}$, where *k* is the normalized exponent of *X*. The function yields the value 111

- $\lfloor X/v \rfloor \cdot v$, when *X* is nonnegative and *Radix_Digits* is positive; 112
- $\lceil X/v \rceil \cdot v$, when *X* is negative and *Radix_Digits* is positive. 113

Constraint_Error is raised when *Radix_Digits* is zero or negative. A zero result, which can only occur when *X* is zero, has the sign of *X*. See A.5.3. 114

115/1 A'Length For a prefix A that is of an array type (after any implicit dereference), or denotes a constrained array subtype:

116 A'Length denotes the number of values of the first index range (zero for a null range); its type is *universal_integer*. See 3.6.2.

117/1 A'Length(N)
For a prefix A that is of an array type (after any implicit dereference), or denotes a constrained array subtype:

118 A'Length(N) denotes the number of values of the N-th index range (zero for a null range); its type is *universal_integer*. See 3.6.2.

119 S'Machine For every subtype S of a floating point type *T*:

120 S'Machine denotes a function with the following specification:

121
```
function S'Machine (X : T)
  return T
```

122 If *X* is a machine number of the type *T*, the function yields *X*; otherwise, it yields the value obtained by rounding or truncating *X* to either one of the adjacent machine numbers of the type *T*. Constraint_Error is raised if rounding or truncating *X* to the precision of the machine numbers results in a value outside the base range of S. A zero result has the sign of *X* when S'Signed_Zeros is True. See A.5.3.

123 S'Machine_Emax
For every subtype S of a floating point type *T*:

124 Yields the largest (most positive) value of *exponent* such that every value expressible in the canonical form (for the type *T*), having a *mantissa* of *T*'Machine_Mantissa digits, is a machine number (see 3.5.7) of the type *T*. This attribute yields a value of the type *universal_integer*. See A.5.3.

125 S'Machine_Emin
For every subtype S of a floating point type *T*:

126 Yields the smallest (most negative) value of *exponent* such that every value expressible in the canonical form (for the type *T*), having a *mantissa* of *T*'Machine_Mantissa digits, is a machine number (see 3.5.7) of the type *T*. This attribute yields a value of the type *universal_integer*. See A.5.3.

127 S'Machine_Mantissa
For every subtype S of a floating point type *T*:

128 Yields the largest value of *p* such that every value expressible in the canonical form (for the type *T*), having a *p*-digit *mantissa* and an *exponent* between *T*'Machine_Emin and *T*'Machine_Emax, is a machine number (see 3.5.7) of the type *T*. This attribute yields a value of the type *universal_integer*. See A.5.3.

129 S'Machine_Overflows
For every subtype S of a floating point type *T*:

130 Yields the value True if overflow and divide-by-zero are detected and reported by raising Constraint_Error for every predefined operation that yields a result of the type *T*; yields the value False otherwise. The value of this attribute is of the predefined type Boolean. See A.5.3.

131 S'Machine_Overflows
For every subtype S of a fixed point type *T*:

132 Yields the value True if overflow and divide-by-zero are detected and reported by raising Constraint_Error for every predefined operation that yields a result of the type *T*; yields the

value False otherwise. The value of this attribute is of the predefined type Boolean. See A.5.4.

S'Machine_Radix 133

For every subtype S of a floating point type *T*:

Yields the radix of the hardware representation of the type *T*. The value of this attribute is of the type *universal_integer*. See A.5.3. 134

S'Machine_Radix 135

For every subtype S of a fixed point type *T*:

Yields the radix of the hardware representation of the type *T*. The value of this attribute is of the type *universal_integer*. See A.5.4. 136

S'Machine_Rounding 136.1/2

For every subtype S of a floating point type *T*:

S'Machine_Rounding denotes a function with the following specification: 136.2/2

```
function S'Machine_Rounding (X : T)                    136.3/2
  return T
```

The function yields the integral value nearest to *X*. If *X* lies exactly halfway between two integers, one of those integers is returned, but which of them is returned is unspecified. A zero result has the sign of *X* when S'Signed_Zeros is True. This function provides access to the rounding behavior which is most efficient on the target processor. See A.5.3. 136.4/2

S'Machine_Rounds 137

For every subtype S of a floating point type *T*:

Yields the value True if rounding is performed on inexact results of every predefined operation that yields a result of the type *T*; yields the value False otherwise. The value of this attribute is of the predefined type Boolean. See A.5.3. 138

S'Machine_Rounds 139

For every subtype S of a fixed point type *T*:

Yields the value True if rounding is performed on inexact results of every predefined operation that yields a result of the type *T*; yields the value False otherwise. The value of this attribute is of the predefined type Boolean. See A.5.4. 140

S'Max For every scalar subtype S: 141

S'Max denotes a function with the following specification: 142

```
function S'Max(Left, Right : S'Base)                   143
  return S'Base
```

The function returns the greater of the values of the two parameters. See 3.5. 144

S'Max_Alignment_For_Allocation 144.1/3

For every subtype S:

Denotes the maximum value for Alignment that can be requested by the implementation via Allocate for an access type whose designated subtype is S. The value of this attribute is of type *universal_integer*. See 13.11.1. 144.2/5

S'Max_Size_In_Storage_Elements 145

For every subtype S:

Denotes the maximum value for Size_In_Storage_Elements that can be requested by the implementation via Allocate for an access type whose designated subtype is S. The value of this attribute is of type *universal_integer*. See 13.11.1. 146/5

S'Min For every scalar subtype S: 147

148 S'Min denotes a function with the following specification:

149
```
function S'Min(Left, Right : S'Base)
  return S'Base
```

150 The function returns the lesser of the values of the two parameters. See 3.5.

150.1/2 S'Mod For every modular subtype S:

150.2/2 S'Mod denotes a function with the following specification:

150.3/2
```
function S'Mod (Arg : universal_integer)
  return S'Base
```

150.4/2 This function returns *Arg* **mod** S'Modulus, as a value of the type of S. See 3.5.4.

151 S'Model For every subtype S of a floating point type *T*:

152 S'Model denotes a function with the following specification:

153
```
function S'Model (X : T)
  return T
```

154 If the Numerics Annex is not supported, the meaning of this attribute is implementation defined; see G.2.2 for the definition that applies to implementations supporting the Numerics Annex. See A.5.3.

155 S'Model_Emin
For every subtype S of a floating point type *T*:

156 If the Numerics Annex is not supported, this attribute yields an implementation defined value that is greater than or equal to the value of *T*'Machine_Emin. See G.2.2 for further requirements that apply to implementations supporting the Numerics Annex. The value of this attribute is of the type *universal_integer*. See A.5.3.

157 S'Model_Epsilon
For every subtype S of a floating point type *T*:

158 Yields the value $T\text{'Machine\_Radix}^{1 - T\text{'Model\_Mantissa}}$. The value of this attribute is of the type *universal_real*. See A.5.3.

159 S'Model_Mantissa
For every subtype S of a floating point type *T*:

160 If the Numerics Annex is not supported, this attribute yields an implementation defined value that is greater than or equal to $\lceil d \cdot \log(10) / \log(T\text{'Machine\_Radix}) \rceil + 1$, where d is the requested decimal precision of *T*, and less than or equal to the value of *T*'Machine_Mantissa. See G.2.2 for further requirements that apply to implementations supporting the Numerics Annex. The value of this attribute is of the type *universal_integer*. See A.5.3.

161 S'Model_Small
For every subtype S of a floating point type *T*:

162 Yields the value $T\text{'Machine\_Radix}^{T\text{'Model\_Emin} - 1}$. The value of this attribute is of the type *universal_real*. See A.5.3.

163 S'Modulus For every modular subtype S:

164 S'Modulus yields the modulus of the type of S, as a value of the type *universal_integer*. See 3.5.4.

164.1/5 S'Object_Size
For every subtype S:

164.2/5 If S is definite, denotes the size (in bits) of a stand-alone aliased object, or a component of subtype S in the absence of an aspect_specification or representation item that specifies the

size of the object or component. If S is indefinite, the meaning is implementation-defined. The value of this attribute is of the type *universal_integer*. See 13.3.

X'Old

For a prefix X that denotes an object of a nonlimited type: 164.3/3

Each X'Old in a postcondition expression that is enabled, other than those that occur in subexpressions that are determined to be unevaluated, denotes a constant that is implicitly declared at the beginning of the subprogram body, entry body, or accept statement. See 6.1.1. 164.4/5

S'Class'Output 165

For every subtype S'Class of a class-wide type *T*'Class:

S'Class'Output denotes a procedure with the following specification: 166

```
procedure S'Class'Output(                                                    167/2
   Stream : not null access Ada.Streams.Root_Stream_Type'Class;
   Item   : in T'Class)
```

First writes the external tag of *Item* to *Stream* (by calling String'Output(*Stream*, Tags.External_Tag(*Item*'Tag)) — see 3.9) and then dispatches to the subprogram denoted by the Output attribute of the specific type identified by the tag. Tag_Error is raised if the tag of Item identifies a type declared at an accessibility level deeper than that of S. See 13.13.2. 168/2

S'Output

For every subtype S of a specific type *T*: 169

S'Output denotes a procedure with the following specification: 170

```
procedure S'Output(                                                          171/2
   Stream : not null access Ada.Streams.Root_Stream_Type'Class;
   Item : in T)
```

S'Output writes the value of *Item* to *Stream*, including any bounds or discriminants. See 13.13.2. 172

X'Overlaps_Storage 172.1/3

For a prefix X that denotes an object:

X'Overlaps_Storage denotes a function with the following specification: 172.2/3

```
function X'Overlaps_Storage (Arg : any_type)                                 172.3/3
  return Boolean
```

The actual parameter shall be a name that denotes an object. The object denoted by the actual parameter can be of any type. This function evaluates the names of the objects involved and returns True if the representation of the object denoted by the actual parameter shares at least one bit with the representation of the object denoted by X; otherwise, it returns False. See 13.3. 172.4/3

X'Parallel_Reduce(Reducer, Initial_Value) 172.5/5

For a prefix X of an array type (after any implicit dereference), or that denotes an iterable container object (see 5.5.1):

X'Parallel_Reduce is a reduction expression that yields a result equivalent to replacing the attribute identifier with Reduce and the prefix of the attribute with the value_sequence: 172.6/5

```
[parallel for Item of X => Item]                                             172.7/5
```

See 4.5.10. 172.8/5

D'Partition_Id 173/1

For a prefix D that denotes a library-level declaration, excepting a declaration of or within a declared-pure library unit:

Denotes a value of the type *universal_integer* that identifies the partition in which D was elaborated. If D denotes the declaration of a remote call interface library unit (see E.2.3) the given partition is the one where the body of D was elaborated. See E.1. 174

175 S'Pos For every discrete subtype S:

176 S'Pos denotes a function with the following specification:

177
```
function S'Pos(Arg : S'Base)
  return universal_integer
```

178 This function returns the position number of the value of *Arg*, as a value of type *universal_integer*. See 3.5.5.

179 R.C'Position
For a component C of a composite, non-array object R:

180/2 If the nondefault bit ordering applies to the composite type, and if a component_clause specifies the placement of C, denotes the value given for the position of the component_clause; otherwise, denotes the same value as R.C'Address – R'Address. The value of this attribute is of the type *universal_integer*. See 13.5.2.

181 S'Pred For every scalar subtype S:

182 S'Pred denotes a function with the following specification:

183
```
function S'Pred(Arg : S'Base)
  return S'Base
```

184 For an enumeration type, the function returns the value whose position number is one less than that of the value of *Arg*; Constraint_Error is raised if there is no such value of the type. For an integer type, the function returns the result of subtracting one from the value of *Arg*. For a fixed point type, the function returns the result of subtracting *small* from the value of *Arg*. For a floating point type, the function returns the machine number (as defined in 3.5.7) immediately below the value of *Arg*; Constraint_Error is raised if there is no such machine number. See 3.5.

184.1/5 S'Preelaborable_Initialization
For a nonformal composite subtype S declared within the visible part of a package or a generic package, or a generic formal private subtype or formal derived subtype:

184.2/5 This attribute is of Boolean type, and its value reflects whether the type of S has preelaborable initialization. See 10.2.1.

184.3/2 P'Priority For a prefix P that denotes a protected object:

184.4/2 Denotes a non-aliased component of the protected object P. This component is of type System.Any_Priority and its value is the priority of P. P'Priority denotes a variable if and only if P denotes a variable. A reference to this attribute shall appear only within the body of P. See D.5.2.

184.5/5 S'Put_Image
For every subtype S of a type T other than *universal_real* or *universal_fixed*:

184.6/5 S'Put_Image denotes a procedure with the following specification:

184.7/5
```
procedure S'Put_Image
   (Buffer : in out
             Ada.Strings.Text_Buffers.Root_Buffer_Type'Class;
    Arg    : in T);
```

184.8/5 The default implementation of S'Put_Image writes (using Wide_Wide_Put) an *image* of the value of *Arg*. See 4.10.

185/1 A'Range For a prefix A that is of an array type (after any implicit dereference), or denotes a constrained array subtype:

186 A'Range is equivalent to the range A'First .. A'Last, except that the prefix A is only evaluated once. See 3.6.2.

187 S'Range For every scalar subtype S:

S'Range is equivalent to the range S'First .. S'Last. See 3.5. 188

A'Range(N) 189/1

For a prefix A that is of an array type (after any implicit dereference), or denotes a constrained array subtype:

A'Range(N) is equivalent to the range A'First(N) .. A'Last(N), except that the prefix A is only evaluated once. See 3.6.2. 190

S'Class'Read 191

For every subtype S'Class of a class-wide type *T*'Class:

S'Class'Read denotes a procedure with the following specification: 192

```
procedure S'Class'Read(
   Stream : not null access Ada.Streams.Root_Stream_Type'Class;
   Item : out T'Class)
```
193/2

Dispatches to the subprogram denoted by the Read attribute of the specific type identified by the tag of Item. See 13.13.2. 194

S'Read For every subtype S of a specific type *T*: 195

S'Read denotes a procedure with the following specification: 196

```
procedure S'Read(
   Stream : not null access Ada.Streams.Root_Stream_Type'Class;
   Item : out T)
```
197/2

S'Read reads the value of *Item* from *Stream*. See 13.13.2. 198

V'Reduce(Reducer, Initial_Value) 198.1/5

For a value_sequence V:

This attribute represents a *reduction expression*, and is in the form of a reduction_attribute_reference. See 4.5.10. 198.2/5

X'Reduce(Reducer, Initial_Value) 198.3/5

For a prefix X of an array type (after any implicit dereference), or that denotes an iterable container object (see 5.5.1):

X'Reduce is a reduction expression that yields a result equivalent to replacing the prefix of the attribute with the value_sequence: 198.4/5

```
[for Item of X => Item]
```
198.5/5

See 4.5.10. 198.6/5

P'Relative_Deadline 198.7/5

For a prefix P that denotes a protected object:

Denotes a non-aliased component of the protected object P. This component is of type Ada.Real_Time.Time_Span and its value is the relative deadline of P. P'Relative_Deadline denotes a variable if and only if P denotes a variable. A reference to this attribute shall appear only within the body of P. See D.5.2. 198.8/5

S'Remainder 199

For every subtype S of a floating point type *T*:

S'Remainder denotes a function with the following specification: 200

```
function S'Remainder (X, Y : T)
  return T
```
201

For nonzero *Y*, let *v* be the value $X - n \cdot Y$, where *n* is the integer nearest to the exact value of X/Y; if $|n - X/Y| = 1/2$, then *n* is chosen to be even. If *v* is a machine number of the type *T*, the function yields *v*; otherwise, it yields zero. Constraint_Error is raised if *Y* is zero. A zero result has the sign of *X* when S'Signed_Zeros is True. See A.5.3. 202

202.1/3 F'Result For a prefix F that denotes a function declaration or an access-to-function type:

202.2/5 Within a postcondition expression for F, denotes the return object of the function call for which the postcondition expression is evaluated. The type of this attribute is that of the result subtype of the function or access-to-function type except within a Post'Class postcondition expression for a function with a controlling result or with a controlling access result; in those cases the type of the attribute is described above as part of the Name Resolution Rules for Post'Class. See 6.1.1.

203 S'Round For every decimal fixed point subtype S:

204 S'Round denotes a function with the following specification:

205
```
function S'Round(X : universal_real)
  return S'Base
```

206 The function returns the value obtained by rounding X (away from 0, if X is midway between two values of the type of S). See 3.5.10.

207 S'Rounding For every subtype S of a floating point type *T*:

208 S'Rounding denotes a function with the following specification:

209
```
function S'Rounding (X : T)
  return T
```

210 The function yields the integral value nearest to *X*, rounding away from zero if *X* lies exactly halfway between two integers. A zero result has the sign of *X* when S'Signed_Zeros is True. See A.5.3.

211 S'Safe_First
For every subtype S of a floating point type *T*:

212 Yields the lower bound of the safe range (see 3.5.7) of the type *T*. If the Numerics Annex is not supported, the value of this attribute is implementation defined; see G.2.2 for the definition that applies to implementations supporting the Numerics Annex. The value of this attribute is of the type *universal_real*. See A.5.3.

213 S'Safe_Last
For every subtype S of a floating point type *T*:

214 Yields the upper bound of the safe range (see 3.5.7) of the type *T*. If the Numerics Annex is not supported, the value of this attribute is implementation defined; see G.2.2 for the definition that applies to implementations supporting the Numerics Annex. The value of this attribute is of the type *universal_real*. See A.5.3.

215 S'Scale For every decimal fixed point subtype S:

216 S'Scale denotes the *scale* of the subtype S, defined as the value N such that S'Delta = 10.0**(–N). The scale indicates the position of the point relative to the rightmost significant digits of values of subtype S. The value of this attribute is of the type *universal_integer*. See 3.5.10.

217 S'Scaling For every subtype S of a floating point type *T*:

218 S'Scaling denotes a function with the following specification:

219
```
function S'Scaling (X : T;
                    Adjustment : universal_integer)
  return T
```

220 Let *v* be the value $X \cdot T'\text{Machine\_Radix}^{Adjustment}$. If *v* is a machine number of the type *T*, or if $|v| \geq T'\text{Model\_Small}$, the function yields *v*; otherwise, it yields either one of the machine numbers of the type *T* adjacent to *v*. Constraint_Error is optionally raised if *v* is outside the base range of S. A zero result has the sign of *X* when S'Signed_Zeros is True. See A.5.3.

S'Signed_Zeros 221

For every subtype S of a floating point type *T*:

Yields the value True if the hardware representation for the type *T* has the capability of representing both positively and negatively signed zeros, these being generated and used by the predefined operations of the type *T* as specified in IEC 559:1989; yields the value False otherwise. The value of this attribute is of the predefined type Boolean. See A.5.3. 222

S'Size For every subtype S: 223

If S is definite, denotes the size (in bits) that the implementation would choose for the following objects of subtype S: 224

- A record component of subtype S when the record type is packed. 225
- The formal parameter of an instance of Unchecked_Conversion that converts from subtype S to some other subtype. 226

If S is indefinite, the meaning is implementation defined. The value of this attribute is of the type *universal_integer*. See 13.3. 227

X'Size For a prefix X that denotes an object: 228/1

Denotes the size in bits of the representation of the object. The value of this attribute is of the type *universal_integer*. See 13.3. 229

S'Small For every fixed point subtype S: 230

S'Small denotes the *small* of the type of S. The value of this attribute is of the type *universal_real*. See 3.5.10. 231

S'Storage_Pool 232

For every access-to-object subtype S:

Denotes the storage pool of the type of S. The type of this attribute is Root_Storage_-Pool'Class. See 13.11. 233

S'Storage_Size 234

For every access-to-object subtype S:

Yields the result of calling Storage_Size(S'Storage_Pool), which is intended to be a measure of the number of storage elements reserved for the pool. The type of this attribute is *universal_integer*. See 13.11. 235

T'Storage_Size 236/1

For a prefix T that denotes a task object (after any implicit dereference):

Denotes the number of storage elements reserved for the task. The value of this attribute is of the type *universal_integer*. The Storage_Size includes the size of the task's stack, if any. The language does not specify whether or not it includes other storage associated with the task (such as the “task control block” used by some implementations.) See 13.3. 237

S'Stream_Size 237.1/2

For every subtype S of an elementary type *T*:

Denotes the number of bits read from or written to a stream by the default implementations of S'Read and S'Write. Hence, the number of stream elements required per item of elementary type *T* is: 237.2/3

```
T'Stream_Size / Ada.Streams.Stream_Element'Size
```
237.3/2

The value of this attribute is of type *universal_integer* and is a multiple of Stream_Element'Size. See 13.13.2. 237.4/2

S'Succ For every scalar subtype S: 238

239 S'Succ denotes a function with the following specification:

240
```
function S'Succ(Arg : S'Base)
  return S'Base
```

241 For an enumeration type, the function returns the value whose position number is one more than that of the value of *Arg*; Constraint_Error is raised if there is no such value of the type. For an integer type, the function returns the result of adding one to the value of *Arg*. For a fixed point type, the function returns the result of adding *small* to the value of *Arg*. For a floating point type, the function returns the machine number (as defined in 3.5.7) immediately above the value of *Arg*; Constraint_Error is raised if there is no such machine number. See 3.5.

242 S'Tag For every subtype S of a tagged type *T* (specific or class-wide):

243 S'Tag denotes the tag of the type *T* (or if *T* is class-wide, the tag of the root type of the corresponding class). The value of this attribute is of type Tag. See 3.9.

244 X'Tag For a prefix X that is of a class-wide tagged type (after any implicit dereference):

245 X'Tag denotes the tag of X. The value of this attribute is of type Tag. See 3.9.

246 T'Terminated
For a prefix T that is of a task type (after any implicit dereference):

247 Yields the value True if the task denoted by T is terminated, and False otherwise. The value of this attribute is of the predefined type Boolean. See 9.9.

248 S'Truncation
For every subtype S of a floating point type *T*:

249 S'Truncation denotes a function with the following specification:

250
```
function S'Truncation (X : T)
  return T
```

251 The function yields the value $\lceil X \rceil$ when *X* is negative, and $\lfloor X \rfloor$ otherwise. A zero result has the sign of *X* when S'Signed_Zeros is True. See A.5.3.

252 S'Unbiased_Rounding
For every subtype S of a floating point type *T*:

253 S'Unbiased_Rounding denotes a function with the following specification:

254
```
function S'Unbiased_Rounding (X : T)
  return T
```

255 The function yields the integral value nearest to *X*, rounding toward the even integer if *X* lies exactly halfway between two integers. A zero result has the sign of *X* when S'Signed_Zeros is True. See A.5.3.

256 X'Unchecked_Access
For a prefix X that denotes an aliased view of an object:

257 All rules and semantics that apply to X'Access (see 3.10.2) apply also to X'Unchecked_Access, except that, for the purposes of accessibility rules and checks, it is as if X were declared immediately within a library package. See 13.10.

258 S'Val For every discrete subtype S:

259 S'Val denotes a function with the following specification:

260
```
function S'Val(Arg : universal_integer)
  return S'Base
```

261 This function returns a value of the type of S whose position number equals the value of *Arg*. See 3.5.5.

X'Valid For a prefix X that denotes a scalar object (after any implicit dereference): 262

Yields True if and only if the object denoted by X is normal, has a valid representation, and then, if the preceding conditions hold, the value of X also satisfies the predicates of the nominal subtype of X. The value of this attribute is of the predefined type Boolean. See 13.9.2. 263/4

S'Value For every scalar subtype S: 264

S'Value denotes a function with the following specification: 265

```
function S'Value(Arg : String)                                    266
  return S'Base
```

This function returns a value given an image of the value as a String, ignoring any leading or trailing spaces. See 3.5. 267

P'Version For a prefix P that statically denotes a program unit: 268/1

Yields a value of the predefined type String that identifies the version of the compilation unit that contains the declaration of the program unit. See E.3. 269

S'Wide_Image 270

For every subtype S of a type T:

S'Wide_Image denotes a function with the following specification: 271/5

```
function S'Wide_Image(Arg : S'Base)                                272/5
  return Wide_String
```

S'Wide_Image calls S'Put_Image passing *Arg* (which will typically store a sequence of character values in a text buffer) and then returns the result of retrieving the contents of that buffer with function Wide_Get. See 4.10. 273/5

X'Wide_Image 273.1/5

For a prefix X of a type T other than *universal_real* or *universal_fixed*:

X'Wide_Image denotes the result of calling function S'Wide_Image with *Arg* being X, where S is the nominal subtype of X. See 4.10. 273.2/5

S'Wide_Value 274

For every scalar subtype S:

S'Wide_Value denotes a function with the following specification: 275

```
function S'Wide_Value(Arg : Wide_String)                           276
  return S'Base
```

This function returns a value given an image of the value as a Wide_String, ignoring any leading or trailing spaces. See 3.5. 277

S'Wide_Wide_Image 277.1/2

For every subtype S of a type T:

S'Wide_Wide_Image denotes a function with the following specification: 277.2/5

```
function S'Wide_Wide_Image(Arg : S'Base)                           277.3/5
  return Wide_Wide_String
```

S'Wide_Wide_Image calls S'Put_Image passing *Arg* (which will typically store a sequence of character values in a text buffer) and then returns the result of retrieving the contents of that buffer with function Wide_Wide_Get. See 4.10. 277.4/5

X'Wide_Wide_Image 277.5/5

For a prefix X of a type T other than *universal_real* or *universal_fixed*:

X'Wide_Wide_Image denotes the result of calling function S'Wide_Wide_Image with *Arg* being X, where S is the nominal subtype of X. See 4.10. 277.6/5

277.7/2 S'Wide_Wide_Value

For every scalar subtype S:

277.8/2 S'Wide_Wide_Value denotes a function with the following specification:

277.9/2
```
function S'Wide_Wide_Value(Arg : Wide_Wide_String)
  return S'Base
```

277.10/2 This function returns a value given an image of the value as a Wide_Wide_String, ignoring any leading or trailing spaces. See 3.5.

277.11/2 S'Wide_Wide_Width

For every scalar subtype S:

277.12/5 S'Wide_Wide_Width denotes the maximum length of a Wide_Wide_String returned by S'Wide_Wide_Image over all values of the subtype S, assuming a default implementation of S'Put_Image. It denotes zero for a subtype that has a null range. Its type is *universal_integer*. See 3.5.

278 S'Wide_Width

For every scalar subtype S:

279/5 S'Wide_Width denotes the maximum length of a Wide_String returned by S'Wide_Image over all values of the subtype S, assuming a default implementation of S'Put_Image. It denotes zero for a subtype that has a null range. Its type is *universal_integer*. See 3.5.

280 S'Width For every scalar subtype S:

281/5 S'Width denotes the maximum length of a String returned by S'Image over all values of the subtype S, assuming a default implementation of S'Put_Image. It denotes zero for a subtype that has a null range. Its type is *universal_integer*. See 3.5.

282 S'Class'Write

For every subtype S'Class of a class-wide type *T*'Class:

283 S'Class'Write denotes a procedure with the following specification:

284/2
```
procedure S'Class'Write(
   Stream : not null access Ada.Streams.Root_Stream_Type'Class;
   Item   : in T'Class)
```

285 Dispatches to the subprogram denoted by the Write attribute of the specific type identified by the tag of Item. See 13.13.2.

286 S'Write For every subtype S of a specific type *T*:

287 S'Write denotes a procedure with the following specification:

288/2
```
procedure S'Write(
   Stream : not null access Ada.Streams.Root_Stream_Type'Class;
   Item : in T)
```

289 S'Write writes the value of *Item* to *Stream*. See 13.13.2.

Annex L
(informative)
Language-Defined Pragmas

This Annex summarizes the definitions given elsewhere of the language-defined pragmas. 1

pragma Admission_Policy (*policy*_identifier); — See D.4.1. 1.1/5

This paragraph was deleted. 2/5

pragma All_Calls_Remote[(*library_unit*_name)]; — See J.15.15. 2.1/5

pragma Assert([Check =>] *boolean*_expression[, [Message =>] *string*_expression]); — See 11.4.2. 2.2/2

pragma Assertion_Policy(*policy*_identifier); — See 11.4.2. 2.3/2

pragma Assertion_Policy(2.4/3
*assertion*_aspect_mark => *policy*_identifier
{, *assertion*_aspect_mark => *policy*_identifier}); — See 11.4.2.

This paragraph was deleted. 3/3

pragma Asynchronous (local_name); — See J.15.13. 3.1/3

This paragraph was deleted. 4/3

pragma Atomic (local_name); — See J.15.8. 4.1/3

This paragraph was deleted. 5/3

pragma Atomic_Components (*array*_local_name); — See J.15.8. 5.1/3

This paragraph was deleted. 6/3

pragma Attach_Handler (*handler*_name, expression); — See J.15.7. 6.1/3

pragma Conflict_Check_Policy (*policy*_identifier[, *policy*_identifier]); — See 9.10.1. 6.2/5

This paragraph was deleted. 7/3

This paragraph was deleted. 8/3

pragma Convention([Convention =>] *convention*_identifier,[Entity =>] local_name); — See J.15.5. 8.1/3

pragma CPU (expression); — See J.15.9. 8.2/3

pragma Default_Storage_Pool (storage_pool_indicator); — See 13.11.3. 8.3/3

pragma Detect_Blocking; — See H.5. 8.4/2

pragma Discard_Names[([On =>] local_name)]; — See C.5. 9

pragma Dispatching_Domain (expression); — See J.15.10. 9.1/3

pragma Elaborate(*library_unit*_name{, *library_unit*_name}); — See 10.2.1. 10

pragma Elaborate_All(*library_unit*_name{, *library_unit*_name}); — See 10.2.1. 11

This paragraph was deleted. 12/5

pragma Elaborate_Body[(*library_unit*_name)]; — See J.15.14. 12.1/5

This paragraph was deleted. 13/3

13.1/3 **pragma** Export(
[Convention =>] *convention*_identifier, [Entity =>] local_name
[, [External_Name =>] *external_name_string*_expression]
[, [Link_Name =>] *link_name_string*_expression]); — See J.15.5.

13.2/5 **pragma** Generate_Deadlines; — See D.2.6.

14/3 *This paragraph was deleted.*

14.1/3 **pragma** Import(
[Convention =>] *convention*_identifier, [Entity =>] local_name
[, [External_Name =>] *external_name_string*_expression]
[, [Link_Name =>] *link_name_string*_expression]); — See J.15.5.

14.2/3 **pragma** Independent (*component*_local_name); — See J.15.8.

14.3/3 **pragma** Independent_Components (local_name); — See J.15.8.

15/3 *This paragraph was deleted.*

15.1/3 **pragma** Inline (name{, name}); — See J.15.1.

16 **pragma** Inspection_Point[(*object*_name {, *object*_name})]; — See H.3.2.

17/3 *This paragraph was deleted.*

17.1/3 **pragma** Interrupt_Handler (*handler*_name); — See J.15.7.

18/3 *This paragraph was deleted.*

18.1/3 **pragma** Interrupt_Priority [(expression);] — See J.15.11.

19 **pragma** Linker_Options(*string*_expression); — See B.1.

20 **pragma** List(identifier); — See 2.8.

21 **pragma** Locking_Policy(*policy*_identifier); — See D.3.

21.1/3 *This paragraph was deleted.*

21.2/3 **pragma** No_Return (*subprogram*_local_name{, *subprogram*_local_name}); — See J.15.2.

22 **pragma** Normalize_Scalars; — See H.1.

23 **pragma** Optimize(identifier); — See 2.8.

24/3 *This paragraph was deleted.*

24.1/3 **pragma** Pack (*first_subtype*_local_name); — See J.15.3.

25 **pragma** Page; — See 2.8.

25.1/2 **pragma** Partition_Elaboration_Policy (*policy*_identifier); — See H.6.

25.2/5 *This paragraph was deleted.*

25.3/5 **pragma** Preelaborable_Initialization(direct_name); — See J.15.14.

26/5 *This paragraph was deleted.*

26.1/5 **pragma** Preelaborate[(*library_unit*_name)]; — See J.15.14.

27/3 *This paragraph was deleted.*

27.1/3 **pragma** Priority (expression); — See J.15.11.

pragma Priority_Specific_Dispatching (*policy_*identifier, *first_priority_*expression, *last_priority_*expression); — See D.2.2. 27.2/2

pragma Profile (*profile_*identifier {, *profile_*pragma_argument_association}); — See 13.12. 27.3/3

This paragraph was deleted. 27.4/3

This paragraph was deleted. 28/5

pragma Pure[(*library_unit_*name)]; — See J.15.14. 28.1/5

pragma Queuing_Policy(*policy_*identifier); — See D.4. 29

This paragraph was deleted. 29.1/3

pragma Relative_Deadline (*relative_deadline_*expression); — See J.15.12. 29.2/3

This paragraph was deleted. 30/5

pragma Remote_Call_Interface[(*library_unit_*name)]; — See J.15.15. 30.1/5

This paragraph was deleted. 31/5

pragma Remote_Types[(*library_unit_*name)]; — See J.15.15. 31.1/5

pragma Restrictions(restriction{, restriction}); — See 13.12. 32

pragma Reviewable; — See H.3.1. 33

This paragraph was deleted. 34/5

pragma Shared_Passive[(*library_unit_*name)]; — See J.15.15. 34.1/5

This paragraph was deleted. 35/3

pragma Storage_Size (expression); — See J.15.4. 35.1/3

pragma Suppress(identifier); — See 11.5. 36

pragma Task_Dispatching_Policy(*policy_*identifier); — See D.2.2. 37

This paragraph was deleted. 37.1/3

pragma Unchecked_Union (*first_subtype_*local_name); — See J.15.6. 37.2/3

pragma Unsuppress(identifier); — See 11.5. 37.3/2

This paragraph was deleted. 38/3

pragma Volatile (local_name); — See J.15.8. 38.1/3

This paragraph was deleted. 39/3

pragma Volatile_Components (*array_*local_name); — See J.15.8. 39.1/3

Wording Changes from Ada 83

Pragmas List, Page, and Optimize are now officially defined in 2.8, “Pragmas”. 39.a

Annex M
(informative)
Summary of Documentation Requirements

{*AI05-0299-1*} {*AI12-0442-1*} The Ada language allows for certain target machine dependences in a controlled manner. Each Ada implementation is required to document many characteristics and properties of the target system. This Reference Manual contains specific documentation requirements. In addition, many characteristics that require documentation are identified throughout this Reference Manual as being implementation defined. Finally, this Reference Manual requires documentation of whether implementation advice is followed. The following subclauses provide summaries of these documentation requirements. 1/5

M.1 Specific Documentation Requirements

{*AI12-0442-1*} In addition to implementation-defined characteristics, each Ada implementation is required to document various properties of the implementation: 1/5

Ramification: Most of the items in this list require documentation only for implementations that conform to Specialized Needs Annexes. 1.a/2

- The behavior of implementations in implementation-defined situations shall be documented — see M.2 for a listing. See 1.1.3(19). 2/2
- The set of values that a user-defined Allocate procedure needs to accept for the Alignment parameter. How the standard storage pool is chosen, and how storage is allocated by standard storage pools. See 13.11(22). 3/2
- The algorithm used for random number generation, including a description of its period. See A.5.2(44). 4/2
- The minimum time interval between calls to the time-dependent Reset procedure that is guaranteed to initiate different random number sequences. See A.5.2(45). 5/2
- The conditions under which Io_Exceptions.Name_Error, Io_Exceptions.Use_Error, and Io_Exceptions.Device_Error are propagated. See A.13(15). 6/2
- The behavior of package Environment_Variables when environment variables are changed by external mechanisms. See A.17(30/2). 7/2
- The overhead of calling machine-code or intrinsic subprograms. See C.1(6). 8/2
- The types and attributes used in machine code insertions. See C.1(7). 9/2
- The subprogram calling conventions for all supported convention identifiers. See C.1(8/3). 10/2
- The mapping between the Link_Name or Ada designator and the external link name. See C.1(9). 11/2
- The treatment of interrupts. See C.3(22). 12/2
- The metrics for interrupt handlers. See C.3.1(16). 13/2
- If the Ceiling_Locking policy is in effect, the default ceiling priority for a protected object that specifies an interrupt handler aspect. See C.3.2(24/5). 14/3
- Any circumstances when the elaboration of a preelaborated package causes code to be executed. See C.4(12). 15/2
- Whether a partition can be restarted without reloading. See C.4(13). 16/2
- The effect of calling Current_Task from an entry body or interrupt handler. See C.7.1(19). 17/2

18/2 • For package Task_Attributes, limits on the number and size of task attributes, and how to configure any limits. See C.7.2(19).

19/2 • The metrics for the Task_Attributes package. See C.7.2(27).

20/2 • The details of the configuration used to generate the values of all metrics. See D(2).

21/2 • The maximum priority inversion a user task can experience from the implementation. See D.2.3(12/2).

22/2 • The amount of time that a task can be preempted for processing on behalf of lower-priority tasks. See D.2.3(13/2).

23/2 • The quantum values supported for round robin dispatching. See D.2.5(16/2).

24/2 • The accuracy of the detection of the exhaustion of the budget of a task for round robin dispatching. See D.2.5(17/2).

25/2 • Any conditions that cause the completion of the setting of the deadline of a task to be delayed for a multiprocessor. See D.2.6(32/2).

26/2 • Any conditions that cause the completion of the setting of the priority of a task to be delayed for a multiprocessor. See D.5.1(12.1/2).

27/2 • The metrics for Set_Priority. See D.5.1(14).

28/2 • The metrics for setting the priority of a protected object. See D.5.2(10).

29/2 • On a multiprocessor, any conditions that cause the completion of an aborted construct to be delayed later than what is specified for a single processor. See D.6(3).

30/2 • The metrics for aborts. See D.6(8).

31/2 • The values of Time_First, Time_Last, Time_Span_First, Time_Span_Last, Time_Span_Unit, and Tick for package Real_Time. See D.8(33).

32/2 • The properties of the underlying time base used in package Real_Time. See D.8(34).

33/2 • Any synchronization of package Real_Time with external time references. See D.8(35).

34/5 • Any aspects of the external environment that can interfere with package Real_Time. See D.8(36/5).

35/2 • The metrics for package Real_Time. See D.8(45).

36/2 • The minimum value of the delay expression of a delay_relative_statement that causes a task to actually be blocked. See D.9(7).

37/2 • The minimum difference between the value of the delay expression of a delay_until_statement and the value of Real_Time.Clock, that causes the task to actually be blocked. See D.9(8).

38/2 • The metrics for delay statements. See D.9(13).

39/2 • The upper bound on the duration of interrupt blocking caused by the implementation. See D.12(5).

40/2 • The metrics for entry-less protected objects. See D.12(12).

41/2 • The values of CPU_Time_First, CPU_Time_Last, CPU_Time_Unit, and CPU_Tick of package Execution_Time. See D.14(21/2).

42/3 • The properties of the mechanism used to implement package Execution_Time, including the values of the constants defined in the package. See D.14(22/2).

43/2 • The metrics for execution time. See D.14(27).

44/2 • The metrics for timing events. See D.15(24).

- The processor(s) on which the clock interrupt is handled; the processors on which each Interrupt_Id can be handled. See D.16.1(32). 44.1/3
- Whether the RPC-receiver is invoked from concurrent tasks, and if so, the number of such tasks. See E.5(25). 45/2
- Any techniques used to reduce cancellation errors in Numerics.Generic_Real_Arrays shall be documented. See G.3.1(86/2). 46/2
- Any techniques used to reduce cancellation errors in Numerics.Generic_Complex_Arrays shall be documented. See G.3.2(155/2). 47/2
- If a pragma Normalize_Scalars applies, the implicit initial values of scalar subtypes shall be documented. Such a value should be an invalid representation when possible; any cases when is it not shall be documented. See H.1(5/2). 48/2
- The range of effects for each bounded error and each unspecified effect. If the effects of a given erroneous construct are constrained, the constraints shall be documented. See H.2(1). 49/2
- For each inspection point, a mapping between each inspectable object and the machine resources where the object's value can be obtained shall be provided. See H.3.2(8). 50/2
- If a pragma Restrictions(No_Exceptions) is specified, the effects of all constructs where language-defined checks are still performed. See H.4(25). 51/2
- The interrupts to which a task entry may be attached. See J.7.1(12). 52/2
- The type of entry call invoked for an interrupt entry. See J.7.1(13). 53/2

M.2 Implementation-Defined Characteristics

{*AI12-0442-1*} The Ada language allows for certain machine dependences in a controlled manner. Each Ada implementation is required to document all implementation-defined characteristics: 1/5

Ramification: It need not document unspecified characteristics. 1.a

Some of the items in this list require documentation only for implementations that conform to Specialized Needs Annexes. 1.b

- Whether or not each recommendation given in Implementation Advice is followed — see M.3 for a listing. See 1.1.2(37). 2/2
- Capacity limitations of the implementation. See 1.1.3(3). 3
- Variations from the standard that are impractical to avoid given the implementation's execution environment. See 1.1.3(6). 4
- Which code_statements cause external interactions. See 1.1.3(10). 5
- The coded representation for the text of an Ada program. See 2.1(4/5). 6
- The semantics of an Ada program whose text is not in Normalization Form C. See 2.1(4.1/5). 6.1/5
- *This paragraph was deleted.* 7/2
- The representation for an end of line. See 2.2(2/3). 8
- Maximum supported line length and lexical element length. See 2.2(14). 9
- Implementation-defined pragmas. See 2.8(14). 10
- Effect of pragma Optimize. See 2.8(27). 11
- The message string associated with the Assertion_Error exception raised by the failure of a predicate check if there is no applicable Predicate_Failure aspect. See 3.2.4(31). 11.1/4
- The predefined integer types declared in Standard. See 3.5.4(25). 13

14 • Any nonstandard integer types and the operators defined for them. See 3.5.4(26/5).

15 • Any nonstandard real types and the operators defined for them. See 3.5.6(8/5).

16 • What combinations of requested decimal precision and range are supported for floating point types. See 3.5.7(7).

17 • The predefined floating point types declared in Standard. See 3.5.7(16/5).

18 • The *small* of an ordinary fixed point type. See 3.5.9(8/2).

19 • What combinations of *small*, range, and *digits* are supported for fixed point types. See 3.5.9(10).

20/2 • The result of Tags.Wide_Wide_Expanded_Name for types declared within an unnamed block_statement. See 3.9(10).

20.1/2 • The sequence of characters of the value returned by Tags.Expanded_Name (respectively, Tags.Wide_Expanded_Name) when some of the graphic characters of Tags.Wide_Wide_Expanded_Name are not defined in Character (respectively, Wide_Character). See 3.9(10.1/2).

21 • Implementation-defined attributes. See 4.1.4(12/5).

21.1/5 • The value of the parameter to Empty for some container_aggregates. See 4.3.5(40).

21.2/5 • The maximum number of chunks for a parallel reduction expression without a chunk_specification. See 4.5.10(21).

21.3/2 • Rounding of real static expressions which are exactly half-way between two machine numbers. See 4.9(38/2).

21.4/5 • The maximum number of chunks for a parallel generalized iterator without a chunk_specification. See 5.5.2(10).

21.5/5 • The number of chunks for an array component iterator. See 5.5.2(11).

21.6/5 • Any extensions of the Global aspect. See 6.1.2(43).

21.7/5 • The circumstances in which the implementation passes in the null value for a view conversion of an access type used as an **out** parameter. See 6.4.1(19/5).

21.8/5 • Any extensions of the Default_Initial_Condition aspect. See 7.3.3(11).

22 • Any implementation-defined time types. See 9.6(6/3).

23 • The time base associated with relative delays. See 9.6(20).

24 • The time base of the type Calendar.Time. See 9.6(23).

25/2 • The time zone used for package Calendar operations. See 9.6(24/2).

26 • Any limit on delay_until_statements of select_statements. See 9.6(29).

26.1/2 • The result of Calendar.Formatting.Image if its argument represents more than 100 hours. See 9.6.1(86/5).

27/3 • *This paragraph was deleted.*

27.1/5 • Implementation-defined conflict check policies. See 9.10.1(5).

28 • The representation for a compilation. See 10.1(2).

29 • Any restrictions on compilations that contain multiple compilation_units. See 10.1(4).

30 • The mechanisms for creating an environment and for adding and replacing compilation units. See 10.1.4(3/2).

30.1/2 • The mechanisms for adding a compilation unit mentioned in a limited_with_clause to an environment. See 10.1.4(3).

- The manner of explicitly assigning library units to a partition. See 10.2(2). 31
- The implementation-defined means, if any, of specifying which compilation units are needed by a given compilation unit. See 10.2(2). 32
- The manner of designating the main subprogram of a partition. See 10.2(7). 33
- The order of elaboration of library_items. See 10.2(18). 34
- Parameter passing and function return for the main subprogram. See 10.2(21). 35
- The mechanisms for building and running partitions. See 10.2(24/5). 36
- The details of program execution, including program termination. See 10.2(25). 37
- The semantics of any nonactive partitions supported by the implementation. See 10.2(28/3). 38
- The information returned by Exception_Message. See 11.4.1(10.1/4). 39
- The result of Exceptions.Wide_Wide_Exception_Name for exceptions declared within an unnamed block_statement. See 11.4.1(12). 40/2
- The sequence of characters of the value returned by Exceptions.Exception_Name (respectively, Exceptions.Wide_Exception_Name) when some of the graphic characters of Exceptions.Wide_Wide_Exception_Name are not defined in Character (respectively, Wide_Character). See 11.4.1(12.1/2). 40.1/2
- The information returned by Exception_Information. See 11.4.1(13/2). 41
- Implementation-defined *policy*_identifiers and *assertion*_aspect_marks allowed in a pragma Assertion_Policy. See 11.4.2(9/5). 41.1/3
- The default assertion policy. See 11.4.2(10). 41.2/2
- Implementation-defined check names. See 11.5(27). 42
- Existence and meaning of second parameter of pragma Unsuppress. See 11.5(27.1/2). 42.1/2
- The cases that cause conflicts between the representation of the ancestors of a type_declaration. See 13.1(13.1/3). 42.2/2
- The interpretation of each representation aspect. See 13.1(20). 43/3
- Any restrictions placed upon the specification of representation aspects. See 13.1(20). 44/3
- Implementation-defined aspects, including the syntax for specifying such aspects and the legality rules for such aspects. See 13.1.1(38). 44.1/3
- The set of machine scalars. See 13.3(8.1/3). 44.2/2
- The meaning of Size for indefinite subtypes. See 13.3(48). 45
- The meaning of Object_Size for indefinite subtypes. See 13.3(58). 45.1/5
- The default external representation for a type tag. See 13.3(75/3). 46
- What determines whether a compilation unit is the same in two different partitions. See 13.3(76). 47
- Implementation-defined components. See 13.5.1(15). 48
- If Word_Size = Storage_Unit, the default bit ordering. See 13.5.3(5). 49
- The contents of the visible part of package System. See 13.7(2). 50/2
- The range of Storage_Elements.Storage_Offset, the modulus of Storage_Elements.Storage_-Element, and the declaration of Storage_Elements.Integer_Address. See 13.7.1(11). 50.1/2
- The contents of the visible part of package System.Machine_Code, and the meaning of code_statements. See 13.8(7). 51

51.1/2 • The result of unchecked conversion for instances with scalar result types whose result is not defined by the language. See 13.9(11).

52/2 • The effect of unchecked conversion for instances with nonscalar result types whose effect is not defined by the language. See 13.9(11).

53/2 • *This paragraph was deleted.*

54 • Whether or not the implementation provides user-accessible names for the standard pool type(s). See 13.11(17).

55/2 • The meaning of Storage_Size when neither the Storage_Size nor the Storage_Pool is specified for an access type. See 13.11(18).

56/2 • *This paragraph was deleted.*

56.1/4 • The effect of specifying aspect Default_Storage_Pool on an instance of a language-defined generic unit. See 13.11.3(5).

57/3 • *This paragraph was deleted.*

57.1/3 • Implementation-defined restrictions allowed in a pragma Restrictions. See 13.12(8.7/3).

58 • The consequences of violating limitations on Restrictions pragmas. See 13.12(9).

58.1/3 • Implementation-defined usage profiles allowed in a pragma Profile. See 13.12(15).

59/2 • The contents of the stream elements read and written by the Read and Write attributes of elementary types. See 13.13.2(9).

60 • The names and characteristics of the numeric subtypes declared in the visible part of package Standard. See A.1(3).

60.1/2 • The values returned by Strings.Hash. See A.4.9(3/2).

60.2/5 • The value returned by a call to a Text_Buffer Get procedure if any character in the returned sequence is not defined in Character. See A.4.12(34).

60.3/5 • The value returned by a call to a Text_Buffer Wide_Get procedure if any character in the returned sequence is not defined in Wide_Character. See A.4.12(34).

61 • The accuracy actually achieved by the elementary functions. See A.5.1(1).

62 • The sign of a zero result from some of the operators or functions in Numerics.Generic_Elementary_Functions, when Float_Type'Signed_Zeros is True. See A.5.1(46).

63 • The value of Numerics.Float_Random.Max_Image_Width. See A.5.2(27).

64 • The value of Numerics.Discrete_Random.Max_Image_Width. See A.5.2(27).

65/2 • *This paragraph was deleted.*

66 • The string representation of a random number generator's state. See A.5.2(38).

67/2 • *This paragraph was deleted.*

68 • The values of the Model_Mantissa, Model_Emin, Model_Epsilon, Model, Safe_First, and Safe_Last attributes, if the Numerics Annex is not supported. See A.5.3(72).

69/2 • *This paragraph was deleted.*

70 • The value of Buffer_Size in Storage_IO. See A.9(10).

71/2 • The external files associated with the standard input, standard output, and standard error files. See A.10(5).

72 • The accuracy of the value produced by Put. See A.10.9(36).

- Current size for a stream file for which positioning is not supported. See A.12.1(1.1/1). 72.1/1
- The meaning of Argument_Count, Argument, and Command_Name for package Command_Line. The bounds of type Command_Line.Exit_Status. See A.15(1). 73/2
- The interpretation of file names and directory names. See A.16(46/2). 73.1/2
- The maximum value for a file size in Directories. See A.16(87/2). 73.2/2
- The result for Directories.Size for a directory or special file. See A.16(93/2). 73.3/2
- The result for Directories.Modification_Time for a directory or special file. See A.16(95/2). 73.4/2
- The interpretation of a nonnull search pattern in Directories. See A.16(104/3). 73.5/2
- The results of a Directories search if the contents of the directory are altered while a search is in progress. See A.16(110/3). 73.6/2
- The definition and meaning of an environment variable. See A.17(1/2). 73.7/2
- The circumstances where an environment variable cannot be defined. See A.17(16/2). 73.8/2
- Environment names for which Set has the effect of Clear. See A.17(17/2). 73.9/2
- The value of Containers.Hash_Type'Modulus. The value of Containers.Count_Type'Last. See A.18.1(7/2). 73.10/2
- Implementation-defined convention names. See B.1(11/3). 74
- The meaning of link names. See B.1(36). 75
- The manner of choosing link names when neither the link name nor the address of an imported or exported entity is specified. See B.1(36). 76
- The effect of pragma Linker_Options. See B.1(37). 77
- The contents of the visible part of package Interfaces and its language-defined descendants. See B.2(1). 78
- Implementation-defined children of package Interfaces. See B.2(11). 79/2
- The definitions of certain types and constants in Interfaces.C. See B.3(41). 79.1/2
- The types Floating, Long_Floating, Binary, Long_Binary, Decimal_Element, and COBOL_Character; and the initializations of the variables Ada_To_COBOL and COBOL_To_Ada, in Interfaces.COBOL. See B.4(50). 80/1
- The types Fortran_Integer, Real, Double_Precision, and Character_Set in Interfaces.Fortran. See B.5(17). 80.1/1
- Implementation-defined intrinsic subprograms. See C.1(1/3). 81/2
- *This paragraph was deleted.* 82/2
- *This paragraph was deleted.* 83/2
- Any restrictions on a protected procedure or its containing type when an aspect Attach_handler or Interrupt_Handler is specified. See C.3.1(17). 83.1/3
- Any other forms of interrupt handler supported by the Attach_Handler and Interrupt_Handler aspects. See C.3.1(19). 83.2/3
- *This paragraph was deleted.* 84/2
- The semantics of some attributes and functions of an entity for which aspect Discard_Names is True. See C.5(7). 85/4
- The modulus and size of Test_and_Set_Flag. See C.6.3(8). 85.1/5

85.2/5 • The value used to represent the set value for Atomic_Test_and_Set. See C.6.3(10).

86 • The result of the Task_Identification.Image attribute. See C.7.1(7).

87/2 • The value of Current_Task when in a protected entry, interrupt handler, or finalization of a task attribute. See C.7.1(17/3).

88/2 • *This paragraph was deleted.*

88.1/1 • Granularity of locking for Task_Attributes. See C.7.2(16/1).

89/2 • *This paragraph was deleted.*

90/2 • *This paragraph was deleted.*

91 • The declarations of Any_Priority and Priority. See D.1(11).

92 • Implementation-defined execution resources. See D.1(15/5).

93 • Whether, on a multiprocessor, a task that is waiting for access to a protected object keeps its processor busy. See D.2.1(3).

94/2 • The effect of implementation-defined execution resources on task dispatching. See D.2.1(9/2).

95/2 • *This paragraph was deleted.*

96/2 • *This paragraph was deleted.*

97/2 • Implementation defined task dispatching policies. See D.2.2(18).

97.1/2 • The value of Default_Quantum in Dispatching.Round_Robin. See D.2.5(4).

98 • Implementation-defined *policy*_identifiers allowed in a pragma Locking_Policy. See D.3(4).

98.1/2 • The locking policy if no Locking_Policy pragma applies to any unit of a partition. See D.3(6).

99 • Default ceiling priorities. See D.3(10/4).

100 • The ceiling of any protected object used internally by the implementation. See D.3(16).

101 • Implementation-defined queuing policies. See D.4(1/5).

101.1/5 • Implementation-defined admission policies. See D.4.1(1).

102/2 • *This paragraph was deleted.*

103 • Any operations that implicitly require heap storage allocation. See D.7(8).

103.1/4 • When restriction No_Dynamic_CPU_Assignment applies to a partition, the processor on which a task with a CPU value of a Not_A_Specific_CPU will execute. See D.7(10).

103.2/2 • When restriction No_Task_Termination applies to a partition, what happens when a task terminates. See D.7(15.1/2).

103.3/2 • The behavior when restriction Max_Storage_At_Blocking is violated. See D.7(17/1).

103.4/2 • The behavior when restriction Max_Asynchronous_Select_Nesting is violated. See D.7(18/1).

103.5/2 • The behavior when restriction Max_Tasks is violated. See D.7(19).

104/2 • Whether the use of pragma Restrictions results in a reduction in program code or data size or execution time. See D.7(20).

105/2 • *This paragraph was deleted.*

106/2 • *This paragraph was deleted.*

106.1/3 • The value of Barrier_Limit'Last in Synchronous_Barriers. See D.10.1(4/3).

106.2/3 • When an aborted task that is waiting on a Synchronous_Barrier is aborted. See D.10.1(13/3).

- *This paragraph was deleted.* 107/2
- The value of Min_Handler_Ceiling in Execution_Time.Group_Budgets. See D.14.2(7/2). 107.1/3
- The value of CPU_Range'Last in System.Multiprocessors. See D.16(4/3). 107.2/3
- The processor on which the environment task executes in the absence of a value for the aspect CPU. See D.16(13/3). 107.3/3
- The means for creating and executing distributed programs. See E(5). 108
- Any events that can result in a partition becoming inaccessible. See E.1(7). 109
- The scheduling policies, treatment of priorities, and management of shared resources between partitions in certain cases. See E.1(11). 110
- *This paragraph was deleted.* 111/1
- Whether the execution of the remote subprogram is immediately aborted as a result of cancellation. See E.4(13). 112
- The range of type System.RPC.Partition_Id. See E.5(14). 112.1/2
- *This paragraph was deleted.* 113/2
- Implementation-defined interfaces in the PCS. See E.5(26). 114
- The values of named numbers in the package Decimal. See F.2(7). 115
- The value of Max_Picture_Length in the package Text_IO.Editing See F.3.3(16). 116
- The value of Max_Picture_Length in the package Wide_Text_IO.Editing See F.3.4(5). 117
- The value of Max_Picture_Length in the package Wide_Wide_Text_IO.Editing See F.3.5(5). 117.1/2
- The accuracy actually achieved by the complex elementary functions and by other complex arithmetic operations. See G.1(1). 118
- The sign of a zero result (or a component thereof) from any operator or function in Numerics.Generic_Complex_Types, when Real'Signed_Zeros is True. See G.1.1(53). 119
- The sign of a zero result (or a component thereof) from any operator or function in Numerics.Generic_Complex_Elementary_Functions, when Complex_Types.Real'Signed_Zeros is True. See G.1.2(45). 120
- Whether the strict mode or the relaxed mode is the default. See G.2(2). 121
- The result interval in certain cases of fixed-to-float conversion. See G.2.1(10). 122
- The result of a floating point arithmetic operation in overflow situations, when the Machine_Overflows attribute of the result type is False. See G.2.1(13). 123
- The result interval for division (or exponentiation by a negative exponent), when the floating point hardware implements division as multiplication by a reciprocal. See G.2.1(16). 124
- The definition of *close result set*, which determines the accuracy of certain fixed point multiplications and divisions. See G.2.3(5). 125
- Conditions on a *universal_real* operand of a fixed point multiplication or division for which the result shall be in the *perfect result set*. See G.2.3(22). 126
- The result of a fixed point arithmetic operation in overflow situations, when the Machine_Overflows attribute of the result type is False. See G.2.3(27). 127
- The result of an elementary function reference in overflow situations, when the Machine_Overflows attribute of the result type is False. See G.2.4(4). 128

129 • The value of the *angle threshold*, within which certain elementary functions, complex arithmetic operations, and complex elementary functions yield results conforming to a maximum relative error bound. See G.2.4(10).

130 • The accuracy of certain elementary functions for parameters beyond the angle threshold. See G.2.4(10).

131 • The result of a complex arithmetic operation or complex elementary function reference in overflow situations, when the Machine_Overflows attribute of the corresponding real type is False. See G.2.6(5).

132 • The accuracy of certain complex arithmetic operations and certain complex elementary functions for parameters (or components thereof) beyond the angle threshold. See G.2.6(8).

132.1/2 • The accuracy requirements for the subprograms Solve, Inverse, Determinant, Eigenvalues and Eigensystem for type Real_Matrix. See G.3.1(81/2).

132.2/2 • The accuracy requirements for the subprograms Solve, Inverse, Determinant, Eigenvalues and Eigensystem for type Complex_Matrix. See G.3.2(149/2).

133/2 • *This paragraph was deleted.*

134/2 • *This paragraph was deleted.*

134.1/5 • The consequences of violating No_Hidden_Indirect_Globals. See H.4(23.9/5).

135/2 • *This paragraph was deleted.*

136/2 • *This paragraph was deleted.*

136.1/2 • Implementation-defined *policy_*identifiers allowed in a pragma Partition_Elaboration_Policy. See H.6(4/2).

M.3 Implementation Advice

1/5 {*AI12-0442-1*} This Reference Manual sometimes gives advice about handling certain target machine dependences. Each Ada implementation is required to document whether that advice is followed:

1.a/2 **Ramification:** Some of the items in this list require documentation only for implementations that conform to Specialized Needs Annexes.

2/2 • Program_Error should be raised when an unsupported Specialized Needs Annex feature is used at run time. See 1.1.3(20).

3/2 • Implementation-defined extensions to the functionality of a language-defined library unit should be provided by adding children to the library unit. See 1.1.3(21).

4/2 • If a bounded error or erroneous execution is detected, Program_Error should be raised. See 1.1.5(12).

5/2 • Implementation-defined pragmas should have no semantic effect for error-free programs. See 2.8(16/3).

6/2 • Implementation-defined pragmas should not make an illegal program legal, unless they complete a declaration or configure the library_items in an environment. See 2.8(19).

7/2 • Long_Integer should be declared in Standard if the target supports 32-bit arithmetic. No other named integer subtypes should be declared in Standard. See 3.5.4(28).

8/2 • For a two's complement target, modular types with a binary modulus up to System.Max_Int*2+2 should be supported. A nonbinary modulus up to Integer'Last should be supported. See 3.5.4(29).

- Program_Error should be raised for the evaluation of S'Pos for an enumeration type, if the value of the operand does not correspond to the internal code for any enumeration literal of the type. See 3.5.5(8). 9/2
- Long_Float should be declared in Standard if the target supports 11 or more digits of precision. No other named float subtypes should be declared in Standard. See 3.5.7(17). 10/2
- Multidimensional arrays should be represented in row-major order, unless the array has convention Fortran. See 3.6.2(11/3). 11/2
- Tags.Internal_Tag should return the tag of a type, if one exists, whose innermost master is a master of the point of the function call. See 3.9(26.1/3). 12/3
- A real static expression with a nonformal type that is not part of a larger static expression should be rounded the same as the target system. See 4.9(38.1/2). 13/2
- For each language-defined private type T, T'Image should generate an image that would be meaningful based only on the relevant public interfaces. See 4.10(56). 13.1/5
- The value of Duration'Small should be no greater than 100 microseconds. See 9.6(30). 14/2
- The time base for delay_relative_statements should be monotonic. See 9.6(31/5). 15/2
- Leap seconds should be supported if the target system supports them. Otherwise, operations in Calendar.Formatting should return results consistent with no leap seconds. See 9.6.1(89/2). 16/2
- *This paragraph was deleted.* 17/5
- A type declared in a preelaborated package should have the same representation in every elaboration of a given version of the package. See 10.2.1(12). 18/2
- Exception_Information should provide information useful for debugging, and should include the Exception_Name and Exception_Message. See 11.4.1(19). 19/2
- Exception_Message by default should be short, provide information useful for debugging, and should not include the Exception_Name. See 11.4.1(19). 20/2
- Code executed for checks that have been suppressed should be minimized. See 11.5(28). 21/2
- The recommended level of support for all representation items should be followed. See 13.1(28/5). 22/2
- Storage allocated to objects of a packed type should be minimized. See 13.2(6). 23/2
- The recommended level of support for the Pack aspect should be followed. See 13.2(9). 24/3
- For an array X, X'Address should point at the first component of the array rather than the array bounds. See 13.3(14). 25/2
- The recommended level of support for the Address attribute should be followed. See 13.3(19). 26/2
- For any tagged specific subtype *S*, *S*'Class'Alignment should equal *S*'Alignment. See 13.3(28). 26.1/3
- The recommended level of support for the Alignment attribute should be followed. See 13.3(35). 27/2
- The Size of an array object should not include its bounds. See 13.3(41.1/2). 28/2
- If the Size of a subtype is nonconfirming and allows for efficient independent addressability, then the Object_Size of the subtype (unless otherwise specified) should equal the Size of the subtype. See 13.3(52). 29/5
- A Size clause on a composite subtype should not affect the internal layout of components. See 13.3(53). 30/2
- The recommended level of support for the Size attribute should be followed. See 13.3(56). 31/2

31.1/5 • An Object_Size clause on a composite type should not affect the internal layout of components. See 13.3(58).

31.2/5 • If S is a definite first subtype for which Object_Size is not specified, S'Object_Size should be the smallest multiple of the storage element size larger than or equal to S'Size that is consistent with the alignment of S. See 13.3(58).

31.3/5 • The recommended level of support for the Object_Size attribute should be followed. See 13.3(58).

31.4/5 • The Size of most objects of a subtype should equal the Object_Size of the subtype. See 13.3(58).

32/2 • The recommended level of support for the Component_Size attribute should be followed. See 13.3(73).

33/2 • The recommended level of support for enumeration_representation_clauses should be followed. See 13.4(10).

34/2 • The recommended level of support for record_representation_clauses should be followed. See 13.5.1(22).

35/2 • If a component is represented using a pointer to the actual data of the component which is contiguous with the rest of the object, then the storage place attributes should reflect the place of the actual data. If a component is allocated discontiguously from the rest of the object, then a warning should be generated upon reference to one of its storage place attributes. See 13.5.2(5).

36/2 • The recommended level of support for the nondefault bit ordering should be followed. See 13.5.3(8).

37/2 • Type System.Address should be a private type. See 13.7(37).

38/2 • Operations in System and its children should reflect the target environment; operations that do not make sense should raise Program_Error. See 13.7.1(16).

39/2 • Since the Size of an array object generally does not include its bounds, the bounds should not be part of the converted data in an instance of Unchecked_Conversion. See 13.9(14/2).

40/2 • There should not be unnecessary runtime checks on the result of an Unchecked_Conversion; the result should be returned by reference when possible. Restrictions on Unchecked_Conversions should be avoided. See 13.9(15).

41/2 • The recommended level of support for Unchecked_Conversion should be followed. See 13.9(17).

42/2 • Any cases in which heap storage is dynamically allocated other than as part of the evaluation of an allocator should be documented. See 13.11(23).

43/2 • A default storage pool for an access-to-constant type should not have overhead to support deallocation of individual objects. See 13.11(24).

44/2 • Usually, a storage pool for an access discriminant or access parameter should be created at the point of an allocator, and be reclaimed when the designated object becomes inaccessible. For other anonymous access types, the pool should be created at the point where the type is elaborated and may have no mechanism for the deallocation of individual objects. See 13.11(25).

45/2 • For a standard storage pool, an instance of Unchecked_Deallocation should actually reclaim the storage. See 13.11.2(17).

45.1/3 • A call on an instance of Unchecked_Deallocation with a nonnull access value should raise Program_Error if the actual access type of the instance is a type for which the Storage_Size has been specified to be zero or is defined by the language to be zero. See 13.11.2(17.1/3).

45.2/5 • Streams.Storage.Bounded.Stream_Type objects should be implemented without implicit pointers or dynamic allocation. See 13.13.1(37).

- If not specified, the value of Stream_Size for an elementary type should be the number of bits that corresponds to the minimum number of stream elements required by the first subtype of the type, rounded up to the nearest factor or multiple of the word size that is also a multiple of the stream element size. See 13.13.2(1.6/2). 46/2
- The recommended level of support for the Stream_Size attribute should be followed. See 13.13.2(1.8/2). 47/2
- If an implementation provides additional named predefined integer types, then the names should end with "Integer". If an implementation provides additional named predefined floating point types, then the names should end with "Float". See A.1(52). 48/2
- Implementation-defined operations on Wide_Character, Wide_String, Wide_Wide_Character, and Wide_Wide_String should be child units of Wide_Characters or Wide_Wide_Characters. See A.3.1(7/3). 49/2
- The string returned by Wide_Characters.Handling.Character_Set_Version should include either "10646:" or "Unicode". See A.3.5(62). 49.1/3
- Bounded string objects should not be implemented by implicit pointers and dynamic allocation. See A.4.4(106). 50/2
- Strings.Hash should be good a hash function, returning a wide spread of values for different string values, and similar strings should rarely return the same value. See A.4.9(12/2). 51/2
- If an implementation supports other string encoding schemes, a child of Ada.Strings similar to UTF_Encoding should be defined. See A.4.11(107/3). 51.1/3
- Bounded buffer objects should be implemented without dynamic allocation. See A.4.12(36). 51.2/5
- Any storage associated with an object of type Generator of the random number packages should be reclaimed on exit from the scope of the object. See A.5.2(46). 52/2
- Each value of Initiator passed to Reset for the random number packages should initiate a distinct sequence of random numbers, or, if that is not possible, be at least a rapidly varying function of the initiator value. See A.5.2(47). 53/2
- Get_Immediate should be implemented with unbuffered input; input should be available immediately; line-editing should be disabled. See A.10.7(23). 54/2
- Package Directories.Information should be provided to retrieve other information about a file. See A.16(124/2). 55/2
- Directories.Start_Search and Directories.Search should raise Name_Error for malformed patterns. See A.16(125). 56/3
- Directories.Rename should be supported at least when both New_Name and Old_Name are simple names and New_Name does not identify an existing external file. See A.16(126/2). 57/2
- Directories.Hierarchical_File_Names should be provided for systems with hierarchical file naming, and should not be provided on other systems. See A.16.1(36/3). 57.1/3
- If the execution environment supports subprocesses, the current environment variables should be used to initialize the environment variables of a subprocess. See A.17(32/2). 58/2
- Changes to the environment variables made outside the control of Environment_Variables should be reflected immediately. See A.17(33/2). 59/2
- Containers.Hash_Type'Modulus should be at least 2**32. Containers.Count_Type'Last should be at least 2**31–1. See A.18.1(8/2). 60/2
- The worst-case time complexity of Element for Containers.Vector should be $O(\log N)$. See A.18.2(256/2). 61/2

62/2 • The worst-case time complexity of Append with Count = 1 when *N* is less than the capacity for Containers.Vector should be *O*(log *N*). See A.18.2(257/2).

63/2 • The worst-case time complexity of Prepend with Count = 1 and Delete_First with Count=1 for Containers.Vectors should be *O*(*N* log *N*). See A.18.2(258/2).

64/2 • The worst-case time complexity of a call on procedure Sort of an instance of Containers.Vectors.Generic_Sorting should be *O*(*N***2), and the average time complexity should be better than *O*(*N***2). See A.18.2(259/2).

65/2 • Containers.Vectors.Generic_Sorting.Sort and Containers.Vectors.Generic_Sorting.Merge should minimize copying of elements. See A.18.2(260/2).

66/2 • Containers.Vectors.Move should not copy elements, and should minimize copying of internal data structures. See A.18.2(261/2).

67/2 • If an exception is propagated from a vector operation, no storage should be lost, nor any elements removed from a vector unless specified by the operation. See A.18.2(262/2).

68/2 • The worst-case time complexity of Element, Insert with Count=1, and Delete with Count=1 for Containers.Doubly_Linked_Lists should be *O*(log *N*). See A.18.3(160/2).

69/2 • A call on procedure Sort of an instance of Containers.Doubly_Linked_Lists.Generic_Sorting should have an average time complexity better than *O*(*N***2) and worst case no worse than *O*(*N***2). See A.18.3(161/2).

70/2 • Containers.Doubly_Linked_Lists.Move should not copy elements, and should minimize copying of internal data structures. See A.18.3(162/2).

71/2 • If an exception is propagated from a list operation, no storage should be lost, nor any elements removed from a list unless specified by the operation. See A.18.3(163/2).

72/2 • Move for a map should not copy elements, and should minimize copying of internal data structures. See A.18.4(83/2).

73/2 • If an exception is propagated from a map operation, no storage should be lost, nor any elements removed from a map unless specified by the operation. See A.18.4(84/2).

74/2 • The average time complexity of Element, Insert, Include, Replace, Delete, Exclude, and Find operations that take a key parameter for Containers.Hashed_Maps should be *O*(log *N*). The average time complexity of the subprograms of Containers.Hashed_Maps that take a cursor parameter should be *O*(1). The average time complexity of Containers.Hashed_Maps.Reserve_Capacity should be *O*(*N*). See A.18.5(62/2).

75/2 • The worst-case time complexity of Element, Insert, Include, Replace, Delete, Exclude, and Find operations that take a key parameter for Containers.Ordered_Maps should be *O*((log *N*)**2) or better. The worst-case time complexity of the subprograms of Containers.Ordered_Maps that take a cursor parameter should be *O*(1). See A.18.6(95/2).

76/2 • Move for sets should not copy elements, and should minimize copying of internal data structures. See A.18.7(104/2).

77/2 • If an exception is propagated from a set operation, no storage should be lost, nor any elements removed from a set unless specified by the operation. See A.18.7(105/2).

78/2 • The average time complexity of the Insert, Include, Replace, Delete, Exclude, and Find operations of Containers.Hashed_Sets that take an element parameter should be *O*(log *N*). The average time complexity of the subprograms of Containers.Hashed_Sets that take a cursor parameter should be *O*(1). The average time complexity of Containers.Hashed_Sets.-Reserve_Capacity should be *O*(*N*). See A.18.8(88/2).

79/2 • The worst-case time complexity of the Insert, Include, Replace, Delete, Exclude, and Find operations of Containers.Ordered_Sets that take an element parameter should be *O*((log *N*)**2).

The worst-case time complexity of the subprograms of Containers.Ordered_Sets that take a cursor parameter should be *O*(1). See A.18.9(116/2).

- The worst-case time complexity of the Element, Parent, First_Child, Last_Child, Next_Sibling, Previous_Sibling, Insert_Child with Count=1, and Delete operations of Containers.Multiway_Trees should be *O*(log *N*). See A.18.10(231/3). 79.1/3
- Containers.Multiway_Trees.Move should not copy elements, and should minimize copying of internal data structures. See A.18.10(232/3). 79.2/3
- If an exception is propagated from a tree operation, no storage should be lost, nor any elements removed from a tree unless specified by the operation. See A.18.10(233/3). 79.3/3
- Move and Swap in Containers.Indefinite_Holders should not copy any elements, and should minimize copying of internal data structures. See A.18.18(73/5). 79.4/5
- If an exception is propagated from a holder operation, no storage should be lost, nor should the element be removed from a holder container unless specified by the operation. See A.18.18(74/3). 79.5/3
- Bounded vector objects should be implemented without implicit pointers or dynamic allocation. See A.18.19(16/3). 79.6/3
- The implementation advice for procedure Move to minimize copying does not apply to bounded vectors. See A.18.19(17/3). 79.7/3
- Bounded list objects should be implemented without implicit pointers or dynamic allocation. See A.18.20(19/3). 79.8/3
- The implementation advice for procedure Move to minimize copying does not apply to bounded lists. See A.18.20(20/3). 79.9/3
- Bounded hashed map objects should be implemented without implicit pointers or dynamic allocation. See A.18.21(21/3). 79.10/3
- The implementation advice for procedure Move to minimize copying does not apply to bounded hashed maps. See A.18.21(22/3). 79.11/3
- Bounded ordered map objects should be implemented without implicit pointers or dynamic allocation. See A.18.22(18/3). 79.12/3
- The implementation advice for procedure Move to minimize copying does not apply to bounded ordered maps. See A.18.22(19/3). 79.13/3
- Bounded hashed set objects should be implemented without implicit pointers or dynamic allocation. See A.18.23(20/3). 79.14/3
- The implementation advice for procedure Move to minimize copying does not apply to bounded hashed sets. See A.18.23(21/3). 79.15/3
- Bounded ordered set objects should be implemented without implicit pointers or dynamic allocation. See A.18.24(17/3). 79.16/3
- The implementation advice for procedure Move to minimize copying does not apply to bounded ordered sets. See A.18.24(18/3). 79.17/3
- Bounded tree objects should be implemented without implicit pointers or dynamic allocation. See A.18.25(19/3). 79.18/3
- The implementation advice for procedure Move to minimize copying does not apply to bounded trees. See A.18.25(20/3). 79.19/3
- Containers.Generic_Array_Sort and Containers.Generic_Constrained_Array_Sort should have an average time complexity better than *O*(*N***2) and worst case no worse than *O*(*N***2). See A.18.26(10/2). 80/2

81/2 • Containers.Generic_Array_Sort and Containers.Generic_Constrained_Array_Sort should minimize copying of elements. See A.18.26(11/2).

81.1/3 • Containers.Generic_Sort should have an average time complexity better than *O*(*N***2) and worst case no worse than *O*(*N***2). See A.18.26(12/3).

81.2/3 • Containers.Generic_Sort should minimize calls to the generic formal Swap. See A.18.26(13/3).

81.3/3 • Bounded queue objects should be implemented without implicit pointers or dynamic allocation. See A.18.29(13/3).

81.4/3 • Bounded priority queue objects should be implemented without implicit pointers or dynamic allocation. See A.18.31(14/3).

81.5/5 • Bounded holder objects should be implemented without dynamic allocation. See A.18.32(15/5).

82/3 • If Export is supported for a language, the main program should be able to be written in that language. Subprograms named "adainit" and "adafinal" should be provided for elaboration and finalization of the environment task. See B.1(39/3).

83/3 • Automatic elaboration of preelaborated packages should be provided when specifying the Export aspect as True is supported. See B.1(40/3).

84/3 • For each supported convention *L* other than Intrinsic, specifying the aspects Import and Export should be supported for objects of *L*-compatible types and for subprograms, and aspect Convention should be supported for *L*-eligible types and for subprograms. See B.1(41/5).

85/2 • If an interface to C, COBOL, or Fortran is provided, the corresponding package or packages described in Annex B, “Interface to Other Languages” should also be provided. See B.2(13/3).

86/2 • The constants nul, wide_nul, char16_nul, and char32_nul in package Interfaces.C should have a representation of zero. See B.3(62.5/3).

87/2 • If C interfacing is supported, the interface correspondences between Ada and C should be supported. See B.3(71).

87.1/5 • If the C implementation supports unsigned long long and long long, unsigned_long_long and long_long should be supported. See B.3(71).

88/2 • If COBOL interfacing is supported, the interface correspondences between Ada and COBOL should be supported. See B.4(98).

89/2 • If Fortran interfacing is supported, the interface correspondences between Ada and Fortran should be supported. See B.5(26).

90/2 • The machine code or intrinsics support should allow access to all operations normally available to assembly language programmers for the target environment. See C.1(3).

91/2 • Interface to assembler should be supported; the default assembler should be associated with the convention identifier Assembler. See C.1(4/3).

92/5 • If an entity is exported to assembly language, then the implementation should allocate it at an addressable location even if not otherwise referenced from the Ada code. A call to a machine code or assembler subprogram should be treated as if it can read or update every object that is specified as exported. See C.1(5).

93/2 • Little or no overhead should be associated with calling intrinsic and machine-code subprograms. See C.1(10).

94/2 • Intrinsic subprograms should be provided to access any machine operations that provide special capabilities or efficiency not normally available. See C.1(16).

- If the Ceiling_Locking policy is not in effect and the target system allows for finer-grained control of interrupt blocking, a means for the application to specify which interrupts are to be blocked during protected actions should be provided. See C.3(28/2). 95/2
- Interrupt handlers should be called directly by the hardware. See C.3.1(20). 96/2
- Violations of any implementation-defined restrictions on interrupt handlers should be detected before run time. See C.3.1(21). 97/2
- If implementation-defined forms of interrupt handler procedures are supported, then for each such form of a handler, a type analogous to Parameterless_Handler should be specified in a child package of Interrupts, with the same operations as in the predefined package Interrupts. See C.3.2(25). 98/2
- Preelaborated packages should be implemented such that little or no code is executed at run time for the elaboration of entities. See C.4(14). 99/2
- If aspect Discard_Names is True for an entity, then the amount of storage used for storing names associated with that entity should be reduced. See C.5(8/4). 100/4
- A load or store of a volatile object whose size is a multiple of System.Storage_Unit and whose alignment is nonzero, should be implemented by accessing exactly the bits of the object and no others. See C.6(22/5). 101/2
- A load or store of an atomic object should be implemented by a single load or store instruction. See C.6(23/2). 102/2
- If the target domain requires deterministic memory use at run time, storage for task attributes should be pre-allocated statically and the number of attributes pre-allocated should be documented. See C.7.2(30). 103/2
- Finalization of task attributes and reclamation of associated storage should be performed as soon as possible after task termination. See C.7.2(30.1/2). 104/2
- Names that end with "_Locking" should be used for implementation-defined locking policies. See D.3(17). 105/2
- Names that end with "_Queuing" should be used for implementation-defined queuing policies. See D.4(16). 106/2
- The abort_statement should not require the task executing the statement to block. See D.6(9). 107/2
- On a multi-processor, the delay associated with aborting a task on another processor should be bounded. See D.6(10). 108/2
- When feasible, specified restrictions should be used to produce a more efficient implementation. See D.7(21). 109/2
- When appropriate, mechanisms to change the value of Tick should be provided. See D.8(47). 110/2
- Calendar.Clock and Real_Time.Clock should be transformations of the same time base. See D.8(48). 111/2
- The "best" time base which exists in the underlying system should be available to the application through Real_Time.Clock. See D.8(49). 112/2
- On a multiprocessor system, each processor should have a separate and disjoint ready queue. See D.13(9). 112.1/3
- When appropriate, implementations should provide configuration mechanisms to change the value of Execution_Time.CPU_Tick. See D.14(29/2). 113/2
- For a timing event, the handler should be executed directly by the real-time clock interrupt mechanism. See D.15(25). 114/2

114.1/5 • Starting a protected action on a protected object statically assigned to a processor should not use busy-waiting. See D.16(16).

114.2/3 • Each dispatching domain should have separate and disjoint ready queues. See D.16.1(31).

115/2 • The PCS should allow for multiple tasks to call the RPC-receiver. See E.5(28).

116/2 • The System.RPC.Write operation should raise Storage_Error if it runs out of space when writing an item. See E.5(29).

117/2 • If COBOL (respectively, C) is supported in the target environment, then interfacing to COBOL (respectively, C) should be supported as specified in Annex B. See F(7/3).

118/2 • Packed decimal should be used as the internal representation for objects of subtype *S* when *S*'Machine_Radix = 10. See F.1(2).

119/2 • If Fortran (respectively, C) is supported in the target environment, then interfacing to Fortran (respectively, C) should be supported as specified in Annex B. See G(7/3).

120/2 • Mixed real and complex operations (as well as pure-imaginary and complex operations) should not be performed by converting the real (resp. pure-imaginary) operand to complex. See G.1.1(56/5).

121/3 • If Real'Signed_Zeros is True for Numerics.Generic_Complex_Types, a rational treatment of the signs of zero results and result components should be provided. See G.1.1(58).

122/3 • If Complex_Types.Real'Signed_Zeros is True for Numerics.Generic_Complex_Elementary_Functions, a rational treatment of the signs of zero results and result components should be provided. See G.1.2(49).

123/2 • For elementary functions, the forward trigonometric functions without a Cycle parameter should not be implemented by calling the corresponding version with a Cycle parameter. Log without a Base parameter should not be implemented by calling Log with a Base parameter. See G.2.4(19).

124/2 • For complex arithmetic, the Compose_From_Polar function without a Cycle parameter should not be implemented by calling Compose_From_Polar with a Cycle parameter. See G.2.6(15).

125/2 • Solve and Inverse for Numerics.Generic_Real_Arrays should be implemented using established techniques such as LU decomposition and the result should be refined by an iteration on the residuals. See G.3.1(88/3).

126/2 • The equality operator should be used to test that a matrix in Numerics.Generic_Real_Arrays is symmetric. See G.3.1(90/2).

126.1/3 • An implementation should minimize the circumstances under which the algorithm used for Numerics.Generic_Real_Arrays.Eigenvalues and Numerics.Generic_Real_Arrays.Eigensystem fails to converge. See G.3.1(91/3).

127/2 • Solve and Inverse for Numerics.Generic_Complex_Arrays should be implemented using established techniques and the result should be refined by an iteration on the residuals. See G.3.2(158/3).

128/2 • The equality and negation operators should be used to test that a matrix is Hermitian. See G.3.2(160/2).

128.1/3 • An implementation should minimize the circumstances under which the algorithm used for Numerics.Generic_Complex_Arrays.Eigenvalues and Numerics.Generic_Complex_Arrays.Eigensystem fails to converge. See G.3.2(160.1/3).

129/2 • Mixed real and complex operations should not be performed by converting the real operand to complex. See G.3.2(161/2).

130/2 • The information produced by pragma Reviewable should be provided in both a human-readable and machine-readable form, and the latter form should be documented. See H.3.1(19).

- Object code listings should be provided both in a symbolic format and in a numeric format. See H.3.1(20). 131/2
- If the partition elaboration policy is Sequential and the Environment task becomes permanently blocked during elaboration, then the partition should be immediately terminated. See H.6(15/3). 132/3
- When applied to a generic unit, a program unit pragma that is not a library unit pragma should apply to each instance of the generic unit for which there is not an overriding pragma applied directly to the instance. See J.15(9/5). 132.1/5

Annex N
(informative)
Glossary

This paragraph was deleted.{*AI95-00437-01*} {*AI12-0443-1*} 1/5

Informal definitions of various terms are now found in 1.3.

Annex P
(informative)
Syntax Summary

This Annex summarizes the complete syntax of the language.

P.1 Syntax Rules

{*AI12-0426-1*} This subclause lists the complete syntax of the language in the order it appears in this Reference Manual. See 1.1.4 for a description of the notation used. 1/5

2.3:
```
identifier ::=
   identifier_start {identifier_start | identifier_extend}
```

2.3:
```
identifier_start ::=
     letter_uppercase
   | letter_lowercase
   | letter_titlecase
   | letter_modifier
   | letter_other
   | number_letter
```

2.3:
```
identifier_extend ::=
     mark_non_spacing
   | mark_spacing_combining
   | number_decimal
   | punctuation_connector
```

2.4:
```
numeric_literal ::= decimal_literal | based_literal
```

2.4.1:
```
decimal_literal ::= numeral [.numeral] [exponent]
```

2.4.1:
```
numeral ::= digit {[underline] digit}
```

2.4.1:
```
exponent ::= E [+] numeral | E – numeral
```

2.4.1:
```
digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

2.4.2:
```
based_literal ::=
   base # based_numeral [.based_numeral] # [exponent]
```

2.4.2:
```
base ::= numeral
```

2.4.2:
```
based_numeral ::=
   extended_digit {[underline] extended_digit}
```

2.4.2:
```
extended_digit ::= digit | A | B | C | D | E | F
```

2.5:
```
character_literal ::= 'graphic_character'
```

2.6:
```
string_literal ::= "{string_element}"
```

2.6:

string_element ::= "" | *non_quotation_mark*_graphic_character

2.7:
comment ::= --{*non_end_of_line*_character}

2.8:
pragma ::=
  pragma identifier [(pragma_argument_association {, pragma_argument_association})];

2.8:
pragma_argument_association ::=
    [*pragma_argument*_identifier =>] name
  | [*pragma_argument*_identifier =>] expression
  | *pragma_argument*_aspect_mark => name
  | *pragma_argument*_aspect_mark => expression

3.1:
basic_declaration ::=
    type_declaration | subtype_declaration
  | object_declaration | number_declaration
  | subprogram_declaration | abstract_subprogram_declaration
  | null_procedure_declaration | expression_function_declaration
  | package_declaration | renaming_declaration
  | exception_declaration | generic_declaration
  | generic_instantiation

3.1:
defining_identifier ::= identifier

3.2.1:
type_declaration ::= full_type_declaration
  | incomplete_type_declaration
  | private_type_declaration
  | private_extension_declaration

3.2.1:
full_type_declaration ::=
    type defining_identifier [known_discriminant_part] **is** type_definition
      [aspect_specification];
  | task_type_declaration
  | protected_type_declaration

3.2.1:
type_definition ::=
    enumeration_type_definition | integer_type_definition
  | real_type_definition | array_type_definition
  | record_type_definition | access_type_definition
  | derived_type_definition | interface_type_definition

3.2.2:
subtype_declaration ::=
  subtype defining_identifier **is** subtype_indication
      [aspect_specification];

3.2.2:
subtype_indication ::= [null_exclusion] subtype_mark [constraint]

3.2.2:
subtype_mark ::= *subtype*_name

3.2.2:
constraint ::= scalar_constraint | composite_constraint

3.2.2:
scalar_constraint ::=
    range_constraint | digits_constraint | delta_constraint

3.2.2:
composite_constraint ::=
    index_constraint | discriminant_constraint

3.3.1:

object_declaration ::=
   defining_identifier_list : [**aliased**] [**constant**] subtype_indication [:= expression]
      [aspect_specification];
 | defining_identifier_list : [**aliased**] [**constant**] access_definition [:= expression]
      [aspect_specification];
 | defining_identifier_list : [**aliased**] [**constant**] array_type_definition [:= expression]
      [aspect_specification];
 | single_task_declaration
 | single_protected_declaration

3.3.1:
defining_identifier_list ::=
 defining_identifier {, defining_identifier}

3.3.2:
number_declaration ::=
    defining_identifier_list : **constant** := *static_*expression;

3.4:
derived_type_definition ::=
   [**abstract**] [**limited**] **new** *parent_*subtype_indication [[**and** interface_list] record_extension_part]

3.5:
range_constraint ::= **range** range

3.5:
range ::= range_attribute_reference
  | simple_expression .. simple_expression

3.5.1:
enumeration_type_definition ::=
  (enumeration_literal_specification {, enumeration_literal_specification})

3.5.1:
enumeration_literal_specification ::=
   defining_identifier | defining_character_literal

3.5.1:
defining_character_literal ::= character_literal

3.5.4:
integer_type_definition ::=
   signed_integer_type_definition | modular_type_definition

3.5.4:
signed_integer_type_definition ::=
   range *static_*simple_expression .. *static_*simple_expression

3.5.4:
modular_type_definition ::= **mod** *static_*expression

3.5.6:
real_type_definition ::=
   floating_point_definition | fixed_point_definition

3.5.7:
floating_point_definition ::=
   digits *static_*expression [real_range_specification]

3.5.7:
real_range_specification ::=
   range *static_*simple_expression .. *static_*simple_expression

3.5.9:
fixed_point_definition ::=
   ordinary_fixed_point_definition | decimal_fixed_point_definition

3.5.9:
ordinary_fixed_point_definition ::=
   delta *static_*expression real_range_specification

3.5.9:
decimal_fixed_point_definition ::=
   delta *static_*expression **digits** *static_*expression [real_range_specification]

3.5.9:
digits_constraint ::=
  digits *static*_simple_expression [range_constraint]

3.6:
array_type_definition ::=
  unconstrained_array_definition | constrained_array_definition

3.6:
unconstrained_array_definition ::=
  array(index_subtype_definition {, index_subtype_definition}) **of** component_definition

3.6:
index_subtype_definition ::= subtype_mark **range** <>

3.6:
constrained_array_definition ::=
  array (discrete_subtype_definition {, discrete_subtype_definition}) **of** component_definition

3.6:
discrete_subtype_definition ::= *discrete*_subtype_indication | range

3.6:
component_definition ::=
  [**aliased**] subtype_indication
| [**aliased**] access_definition

3.6.1:
index_constraint ::= (discrete_range {, discrete_range})

3.6.1:
discrete_range ::= *discrete*_subtype_indication | range

3.7:
discriminant_part ::= unknown_discriminant_part | known_discriminant_part

3.7:
unknown_discriminant_part ::= (<>)

3.7:
known_discriminant_part ::=
  (discriminant_specification {; discriminant_specification})

3.7:
discriminant_specification ::=
  defining_identifier_list : [null_exclusion] subtype_mark [:= default_expression]
    [aspect_specification]
| defining_identifier_list : access_definition [:= default_expression]
    [aspect_specification]

3.7:
default_expression ::= expression

3.7.1:
discriminant_constraint ::=
  (discriminant_association {, discriminant_association})

3.7.1:
discriminant_association ::=
  [*discriminant*_selector_name {'|' *discriminant*_selector_name} =>] expression

3.8:
record_type_definition ::= [[**abstract**] **tagged**] [**limited**] record_definition

3.8:
record_definition ::=
   record
     component_list
   end record [*record*_identifier]
 | **null record**

3.8:
component_list ::=
    component_item {component_item}

```
  | {component_item} variant_part
  | null;
```

3.8:
```
component_item ::= component_declaration | aspect_clause
```

3.8:
```
component_declaration ::=
  defining_identifier_list : component_definition [:= default_expression]
     [aspect_specification];
```

3.8.1:
```
variant_part ::=
  case discriminant_direct_name is
     variant
    {variant}
  end case;
```

3.8.1:
```
variant ::=
  when discrete_choice_list =>
    component_list
```

3.8.1:
```
discrete_choice_list ::= discrete_choice {'|' discrete_choice}
```

3.8.1:
```
discrete_choice ::=
  choice_expression | discrete_subtype_indication | range | others
```

3.9.1:
```
record_extension_part ::= with record_definition
```

3.9.3:
```
abstract_subprogram_declaration ::=
   [overriding_indicator]
   subprogram_specification is abstract
      [aspect_specification];
```

3.9.4:
```
interface_type_definition ::=
   [limited | task | protected | synchronized] interface [and interface_list]
```

3.9.4:
```
interface_list ::= interface_subtype_mark {and interface_subtype_mark}
```

3.10:
```
access_type_definition ::=
   [null_exclusion] access_to_object_definition
 | [null_exclusion] access_to_subprogram_definition
```

3.10:
```
access_to_object_definition ::=
   access [general_access_modifier] subtype_indication
```

3.10:
```
general_access_modifier ::= all | constant
```

3.10:
```
access_to_subprogram_definition ::=
   access [protected] procedure parameter_profile
 | access [protected] function  parameter_and_result_profile
```

3.10:
```
null_exclusion ::= not null
```

3.10:
```
access_definition ::=
   [null_exclusion] access [constant] subtype_mark
 | [null_exclusion] access [protected] procedure parameter_profile
 | [null_exclusion] access [protected] function parameter_and_result_profile
```

3.10.1:
```
incomplete_type_declaration ::=
```

type defining_identifier [discriminant_part] [**is tagged**];

3.11:
declarative_part ::= {declarative_item}

3.11:
declarative_item ::=
basic_declarative_item | body

3.11:
basic_declarative_item ::=
basic_declaration | aspect_clause | use_clause

3.11:
body ::= proper_body | body_stub

3.11:
proper_body ::=
subprogram_body | package_body | task_body | protected_body

4.1:
name ::=
direct_name | explicit_dereference
| indexed_component | slice
| selected_component | attribute_reference
| type_conversion | function_call
| character_literal | qualified_expression
| generalized_reference | generalized_indexing
| target_name

4.1:
direct_name ::= identifier | operator_symbol

4.1:
prefix ::= name | implicit_dereference

4.1:
explicit_dereference ::= name.**all**

4.1:
implicit_dereference ::= name

4.1.1:
indexed_component ::= prefix(expression {, expression})

4.1.2:
slice ::= prefix(discrete_range)

4.1.3:
selected_component ::= prefix . selector_name

4.1.3:
selector_name ::= identifier | character_literal | operator_symbol

4.1.4:
attribute_reference ::=
prefix'attribute_designator
| reduction_attribute_reference

4.1.4:
attribute_designator ::=
identifier[(*static*_expression)]
| Access | Delta | Digits | Mod

4.1.4:
range_attribute_reference ::= prefix'range_attribute_designator

4.1.4:
range_attribute_designator ::= Range[(*static*_expression)]

4.1.5:
generalized_reference ::= *reference_object*_name

4.1.6:
generalized_indexing ::= *indexable_container_object*_prefix actual_parameter_part

4.3:
aggregate ::=
   record_aggregate | extension_aggregate | array_aggregate
 | delta_aggregate | container_aggregate

4.3.1:
record_aggregate ::= (record_component_association_list)

4.3.1:
record_component_association_list ::=
   record_component_association {, record_component_association}
 | **null record**

4.3.1:
record_component_association ::=
   [component_choice_list =>] expression
 | component_choice_list => <>

4.3.1:
component_choice_list ::=
    *component*_selector_name {'|' *component*_selector_name}
 | **others**

4.3.2:
extension_aggregate ::=
   (ancestor_part **with** record_component_association_list)

4.3.2:
ancestor_part ::= expression | subtype_mark

4.3.3:
array_aggregate ::=
   positional_array_aggregate | null_array_aggregate | named_array_aggregate

4.3.3:
positional_array_aggregate ::=
   (expression, expression {, expression})
 | (expression {, expression}, **others** => expression)
 | (expression {, expression}, **others** => <>)
 | '[' expression {, expression}[, **others** => expression] ']'
 | '[' expression {, expression}, **others** => <> ']'

4.3.3:
null_array_aggregate ::= '[' ']'

4.3.3:
named_array_aggregate ::=
   (array_component_association_list)
 | '[' array_component_association_list ']'

4.3.3:
array_component_association_list ::=
   array_component_association {, array_component_association}

4.3.3:
array_component_association ::=
   discrete_choice_list => expression
 | discrete_choice_list => <>
 | iterated_component_association

4.3.3:
iterated_component_association ::=
   for defining_identifier **in** discrete_choice_list => expression
 | **for** iterator_specification => expression

4.3.4:
delta_aggregate ::= record_delta_aggregate | array_delta_aggregate

4.3.4:
record_delta_aggregate ::=
   (*base*_expression **with delta** record_component_association_list)

4.3.4:

array_delta_aggregate ::=
   (*base*_expression **with delta** array_component_association_list)
 | '[' *base*_expression **with delta** array_component_association_list ']'

4.3.5:
container_aggregate ::=
   null_container_aggregate
 | positional_container_aggregate
 | named_container_aggregate

4.3.5:
null_container_aggregate ::= '[' ']'

4.3.5:
positional_container_aggregate ::= '[' expression{, expression} ']'

4.3.5:
named_container_aggregate ::= '[' container_element_association_list ']'

4.3.5:
container_element_association_list ::=
   container_element_association {, container_element_association}

4.3.5:
container_element_association ::=
   key_choice_list => expression
 | key_choice_list => <>
 | iterated_element_association

4.3.5:
key_choice_list ::= key_choice {'|' key_choice}

4.3.5:
key_choice ::= *key*_expression | discrete_range

4.3.5:
iterated_element_association ::=
   for loop_parameter_specification[**use** *key*_expression] => expression
 | **for** iterator_specification[**use** *key*_expression] => expression

4.4:
expression ::=
    relation {**and** relation} | relation {**and then** relation}
  | relation {**or** relation} | relation {**or else** relation}
  | relation {**xor** relation}

4.4:
choice_expression ::=
    choice_relation {**and** choice_relation}
  | choice_relation {**or** choice_relation}
  | choice_relation {**xor** choice_relation}
  | choice_relation {**and then** choice_relation}
  | choice_relation {**or else** choice_relation}

4.4:
choice_relation ::=
    simple_expression [relational_operator simple_expression]

4.4:
relation ::=
    simple_expression [relational_operator simple_expression]
  | *tested*_simple_expression [**not**] **in** membership_choice_list
  | raise_expression

4.4:
membership_choice_list ::= membership_choice {'|' membership_choice}

4.4:
membership_choice ::= *choice*_simple_expression | range | subtype_mark

4.4:
simple_expression ::=
   [unary_adding_operator] term {binary_adding_operator term}

4.4:
term ::= factor {multiplying_operator factor}

4.4:
factor ::= primary [** primary] | **abs** primary | **not** primary

4.4:
primary ::=
   numeric_literal | **null** | string_literal | aggregate
 | name | allocator | (expression)
 | (conditional_expression) | (quantified_expression)
 | (declare_expression)

4.5:
logical_operator ::= **and** | **or** | **xor**

4.5:
relational_operator ::= = | /= | < | <= | > | >=

4.5:
binary_adding_operator ::= + | – | &

4.5:
unary_adding_operator ::= + | –

4.5:
multiplying_operator ::= * | / | **mod** | **rem**

4.5:
highest_precedence_operator ::= ** | **abs** | **not**

4.5.7:
conditional_expression ::= if_expression | case_expression

4.5.7:
if_expression ::=
  if condition **then** *dependent_*expression
  {**elsif** condition **then** *dependent_*expression}
  [**else** *dependent_*expression]

4.5.7:
condition ::= *boolean_*expression

4.5.7:
case_expression ::=
   case *selecting_*expression **is**
   case_expression_alternative {,
   case_expression_alternative}

4.5.7:
case_expression_alternative ::=
   when discrete_choice_list =>
      *dependent_*expression

4.5.8:
quantified_expression ::=
   for quantifier loop_parameter_specification => predicate
 | **for** quantifier iterator_specification => predicate

4.5.8:
quantifier ::= **all** | **some**

4.5.8:
predicate ::= *boolean_*expression

4.5.9:
declare_expression ::=
   declare {declare_item}
   begin *body_*expression

4.5.9:
declare_item ::= object_declaration | object_renaming_declaration

4.5.10:
reduction_attribute_reference ::=

value_sequence'reduction_attribute_designator
| prefix'reduction_attribute_designator

4.5.10:
value_sequence ::=
'[' [**parallel**[(chunk_specification)] [aspect_specification]]
iterated_element_association ']'

4.5.10:
reduction_attribute_designator ::=
*reduction_*identifier(reduction_specification)

4.5.10:
reduction_specification ::= *reducer_*name, *initial_value_*expression

4.6:
type_conversion ::=
subtype_mark(expression)
| subtype_mark(name)

4.7:
qualified_expression ::=
subtype_mark'(expression) | subtype_mark'aggregate

4.8:
allocator ::=
new [subpool_specification] subtype_indication
| **new** [subpool_specification] qualified_expression

4.8:
subpool_specification ::= (*subpool_handle_*name)

5.1:
sequence_of_statements ::= statement {statement} {label}

5.1:
statement ::=
{label} simple_statement | {label} compound_statement

5.1:
simple_statement ::= null_statement
| assignment_statement | exit_statement
| goto_statement | procedure_call_statement
| simple_return_statement | entry_call_statement
| requeue_statement | delay_statement
| abort_statement | raise_statement
| code_statement

5.1:
compound_statement ::=
if_statement | case_statement
| loop_statement | block_statement
| extended_return_statement
| parallel_block_statement
| accept_statement | select_statement

5.1:
null_statement ::= **null**;

5.1:
label ::= <<*label_*statement_identifier>>

5.1:
statement_identifier ::= direct_name

5.2:
assignment_statement ::=
*variable_*name := expression;

5.2.1:
target_name ::= @

5.3:
if_statement ::=

```
   if condition then
     sequence_of_statements
  {elsif condition then
     sequence_of_statements}
  [else
     sequence_of_statements]
   end if;
```

5.4:

```
case_statement ::=
  case selecting_expression is
     case_statement_alternative
    {case_statement_alternative}
  end case;
```

5.4:

```
case_statement_alternative ::=
  when discrete_choice_list =>
    sequence_of_statements
```

5.5:

```
loop_statement ::=
  [loop_statement_identifier:]
    [iteration_scheme] loop
       sequence_of_statements
     end loop [loop_identifier];
```

5.5:

```
iteration_scheme ::= while condition
  | for loop_parameter_specification
  | for iterator_specification
  | [parallel [aspect_specification]]
    for procedural_iterator
  | parallel [(chunk_specification)] [aspect_specification]
    for loop_parameter_specification
  | parallel [(chunk_specification)] [aspect_specification]
    for iterator_specification
```

5.5:

```
chunk_specification ::=
    integer_simple_expression
  | defining_identifier in discrete_subtype_definition
```

5.5:

```
loop_parameter_specification ::=
  defining_identifier in [reverse] discrete_subtype_definition
    [iterator_filter]
```

5.5:

```
iterator_filter ::= when condition
```

5.5.2:

```
iterator_specification ::=
    defining_identifier [: loop_parameter_subtype_indication] in [reverse] iterator_name
      [iterator_filter]
  | defining_identifier [: loop_parameter_subtype_indication] of [reverse] iterable_name
      [iterator_filter]
```

5.5.2:

```
loop_parameter_subtype_indication ::= subtype_indication | access_definition
```

5.5.3:

```
procedural_iterator ::=
    iterator_parameter_specification of iterator_procedure_call
      [iterator_filter]
```

5.5.3:

```
iterator_parameter_specification ::=
    formal_part
  | (defining_identifier{, defining_identifier})
```

5.5.3:
iterator_procedure_call ::=
    *procedure*_name
  | *procedure*_prefix iterator_actual_parameter_part

5.5.3:
iterator_actual_parameter_part ::=
    (iterator_parameter_association {, iterator_parameter_association})

5.5.3:
iterator_parameter_association ::=
    parameter_association
  | parameter_association_with_box

5.5.3:
parameter_association_with_box ::=
  [*formal_parameter*_selector_name =>] <>

5.6:
block_statement ::=
  [*block*_statement_identifier:]
     [**declare**
        declarative_part]
      begin
        handled_sequence_of_statements
      end [*block*_identifier];

5.6.1:
parallel_block_statement ::=
   parallel [(chunk_specification)] [aspect_specification] **do**
     sequence_of_statements
   and
     sequence_of_statements
  {**and**
     sequence_of_statements}
   end do;

5.7:
exit_statement ::=
  exit [*loop*_name] [**when** condition];

5.8:
goto_statement ::= **goto** *label*_name;

6.1:
subprogram_declaration ::=
   [overriding_indicator]
   subprogram_specification
      [aspect_specification];

6.1:
subprogram_specification ::=
   procedure_specification
 | function_specification

6.1:
procedure_specification ::=
   procedure defining_program_unit_name parameter_profile

6.1:
function_specification ::=
   function defining_designator parameter_and_result_profile

6.1:
designator ::= [parent_unit_name .]identifier | operator_symbol

6.1:
defining_designator ::=
   defining_program_unit_name | defining_operator_symbol

6.1:
defining_program_unit_name ::= [parent_unit_name .]defining_identifier

6.1:
operator_symbol ::= string_literal

6.1:
defining_operator_symbol ::= operator_symbol

6.1:
parameter_profile ::= [formal_part]

6.1:
parameter_and_result_profile ::=
    [formal_part] **return** [null_exclusion] subtype_mark
  | [formal_part] **return** access_definition

6.1:
formal_part ::=
  (parameter_specification {; parameter_specification})

6.1:
parameter_specification ::=
    defining_identifier_list : [**aliased**] mode [null_exclusion] subtype_mark [:= default_expression]
       [aspect_specification]
  | defining_identifier_list : access_definition [:= default_expression]
       [aspect_specification]

6.1:
mode ::= [**in**] | **in out** | **out**

6.1.2:
global_aspect_definition ::=
    null
  | Unspecified
  | global_mode global_designator
  | (global_aspect_element{; global_aspect_element})

6.1.2:
global_aspect_element ::=
    global_mode global_set
  | global_mode **all**
  | global_mode **synchronized**

6.1.2:
global_mode ::=
    basic_global_mode
  | extended_global_mode

6.1.2:
basic_global_mode ::= **in** | **in out** | **out**

6.1.2:
global_set ::= global_name {, global_name}

6.1.2:
global_designator ::= **all** | **synchronized** | global_name

6.1.2:
global_name ::= *object*_name | *package*_name

6.3:
subprogram_body ::=
    [overriding_indicator]
    subprogram_specification
       [aspect_specification] **is**
       declarative_part
    begin
       handled_sequence_of_statements
    end [designator];

6.4:
procedure_call_statement ::=
    *procedure*_name;
  | *procedure*_prefix actual_parameter_part;

6.4:
function_call ::=
  *function*_name
 |*function*_prefix actual_parameter_part

6.4:
actual_parameter_part ::=
  (parameter_association {, parameter_association})

6.4:
parameter_association ::=
  [*formal_parameter*_selector_name =>] explicit_actual_parameter

6.4:
explicit_actual_parameter ::= expression | *variable*_name

6.5:
simple_return_statement ::= **return** [expression];

6.5:
extended_return_object_declaration ::=
   defining_identifier : [**aliased**] [**constant**] return_subtype_indication [:= expression]
      [aspect_specification]

6.5:
extended_return_statement ::=
   return extended_return_object_declaration [**do**
      handled_sequence_of_statements
   end return];

6.5:
return_subtype_indication ::= subtype_indication | access_definition

6.7:
null_procedure_declaration ::=
  [overriding_indicator]
  procedure_specification **is null**
     [aspect_specification];

6.8:
expression_function_declaration ::=
  [overriding_indicator]
  function_specification **is**
     (expression)
     [aspect_specification];
 | [overriding_indicator]
  function_specification **is**
     aggregate
     [aspect_specification];

7.1:
package_declaration ::= package_specification;

7.1:
package_specification ::=
   package defining_program_unit_name
      [aspect_specification] **is**
     {basic_declarative_item}
  [**private**
     {basic_declarative_item}]
   end [[parent_unit_name.]identifier]

7.2:
package_body ::=
   package body defining_program_unit_name
      [aspect_specification] **is**
     declarative_part
  [**begin**
      handled_sequence_of_statements]
   end [[parent_unit_name.]identifier];

7.3:
private_type_declaration ::=
  type defining_identifier [discriminant_part] **is** [[**abstract**] **tagged**] [**limited**] **private**
    [aspect_specification];

7.3:
private_extension_declaration ::=
  type defining_identifier [discriminant_part] **is**
    [**abstract**] [**limited** | **synchronized**] **new** *ancestor_*subtype_indication
    [**and** interface_list] **with private**
      [aspect_specification];

8.3.1:
overriding_indicator ::= [**not**] **overriding**

8.4:
use_clause ::= use_package_clause | use_type_clause

8.4:
use_package_clause ::= **use** *package_*name {, *package_*name};

8.4:
use_type_clause ::= **use** [**all**] **type** subtype_mark {, subtype_mark};

8.5:
renaming_declaration ::=
    object_renaming_declaration
  | exception_renaming_declaration
  | package_renaming_declaration
  | subprogram_renaming_declaration
  | generic_renaming_declaration

8.5.1:
object_renaming_declaration ::=
   defining_identifier [: [null_exclusion] subtype_mark] **renames** *object_*name
      [aspect_specification];
 | defining_identifier : access_definition **renames** *object_*name
      [aspect_specification];

8.5.2:
exception_renaming_declaration ::=
   defining_identifier : **exception renames** *exception_*name
     [aspect_specification];

8.5.3:
package_renaming_declaration ::=
   package defining_program_unit_name **renames** *package_*name
     [aspect_specification];

8.5.4:
subprogram_renaming_declaration ::=
   [overriding_indicator]
   subprogram_specification **renames** *callable_entity_*name
      [aspect_specification];

8.5.5:
generic_renaming_declaration ::=
   generic package defining_program_unit_name **renames** *generic_package_*name
      [aspect_specification];
 | **generic procedure** defining_program_unit_name **renames** *generic_procedure_*name
      [aspect_specification];
 | **generic function** defining_program_unit_name **renames** *generic_function_*name
      [aspect_specification];

9.1:
task_type_declaration ::=
  task type defining_identifier [known_discriminant_part]
      [aspect_specification] [**is**
    [**new** interface_list **with**]
    task_definition];

9.1:
single_task_declaration ::=
  task defining_identifier
      [aspect_specification] [**is**
    [**new** interface_list **with**]
    task_definition];

9.1:
task_definition ::=
    {task_item}
 [**private**
    {task_item}]
 end [*task*_identifier]

9.1:
task_item ::= entry_declaration | aspect_clause

9.1:
task_body ::=
  task body defining_identifier
      [aspect_specification] **is**
    declarative_part
  begin
    handled_sequence_of_statements
  end [*task*_identifier];

9.4:
protected_type_declaration ::=
 protected type defining_identifier [known_discriminant_part]
      [aspect_specification] **is**
    [**new** interface_list **with**]
    protected_definition;

9.4:
single_protected_declaration ::=
 protected defining_identifier
      [aspect_specification] **is**
    [**new** interface_list **with**]
    protected_definition;

9.4:
protected_definition ::=
   { protected_operation_declaration }
[**private**
   { protected_element_declaration }]
 end [*protected*_identifier]

9.4:
protected_operation_declaration ::= subprogram_declaration
    | entry_declaration
    | aspect_clause

9.4:
protected_element_declaration ::= protected_operation_declaration
    | component_declaration

9.4:
protected_body ::=
 protected body defining_identifier
      [aspect_specification] **is**
  { protected_operation_item }
 end [*protected*_identifier];

9.4:
protected_operation_item ::= subprogram_declaration
    | subprogram_body
    | null_procedure_declaration
    | expression_function_declaration
    | entry_body
    | aspect_clause

9.5:
synchronization_kind ::= By_Entry | By_Protected_Procedure | Optional

9.5.2:
entry_declaration ::=
  [overriding_indicator]
  entry defining_identifier [(discrete_subtype_definition)] parameter_profile
    [aspect_specification];

9.5.2:
accept_statement ::=
  accept *entry*_direct_name [(entry_index)] parameter_profile [**do**
    handled_sequence_of_statements
  end [*entry*_identifier]];

9.5.2:
entry_index ::= expression

9.5.2:
entry_body ::=
   entry defining_identifier entry_body_formal_part
     [aspect_specification]
   entry_barrier **is**
     declarative_part
   begin
     handled_sequence_of_statements
   end [*entry*_identifier];

9.5.2:
entry_body_formal_part ::= [(entry_index_specification)] parameter_profile

9.5.2:
entry_barrier ::= **when** condition

9.5.2:
entry_index_specification ::=
   for defining_identifier **in** discrete_subtype_definition [aspect_specification]

9.5.3:
entry_call_statement ::= *entry*_name [actual_parameter_part];

9.5.4:
requeue_statement ::= **requeue** *procedure_or_entry*_name [**with abort**];

9.6:
delay_statement ::= delay_until_statement | delay_relative_statement

9.6:
delay_until_statement ::= **delay until** *delay*_expression;

9.6:
delay_relative_statement ::= **delay** *delay*_expression;

9.7:
select_statement ::=
  selective_accept
 | timed_entry_call
 | conditional_entry_call
 | asynchronous_select

9.7.1:
selective_accept ::=
 select
  [guard]
   select_alternative
{ **or**
  [guard]
   select_alternative }
[**else**
  sequence_of_statements]
 end select;

9.7.1:
guard ::= **when** condition =>

9.7.1:
select_alternative ::=
  accept_alternative
 | delay_alternative
 | terminate_alternative

9.7.1:
accept_alternative ::=
 accept_statement [sequence_of_statements]

9.7.1:
delay_alternative ::=
 delay_statement [sequence_of_statements]

9.7.1:
terminate_alternative ::= **terminate**;

9.7.2:
timed_entry_call ::=
 select
  entry_call_alternative
 or
  delay_alternative
 end select;

9.7.2:
entry_call_alternative ::=
 procedure_or_entry_call [sequence_of_statements]

9.7.2:
procedure_or_entry_call ::=
 procedure_call_statement | entry_call_statement

9.7.3:
conditional_entry_call ::=
 select
  entry_call_alternative
 else
  sequence_of_statements
 end select;

9.7.4:
asynchronous_select ::=
 select
  triggering_alternative
 then abort
  abortable_part
 end select;

9.7.4:
triggering_alternative ::= triggering_statement [sequence_of_statements]

9.7.4:
triggering_statement ::= procedure_or_entry_call | delay_statement

9.7.4:
abortable_part ::= sequence_of_statements

9.8:
abort_statement ::= **abort** *task_*name {, *task_*name};

10.1.1:
compilation ::= {compilation_unit}

10.1.1:
compilation_unit ::=
   context_clause library_item
 | context_clause subunit

10.1.1:

library_item ::= [**private**] library_unit_declaration
  | library_unit_body
  | [**private**] library_unit_renaming_declaration

10.1.1:
library_unit_declaration ::=
    subprogram_declaration | package_declaration
  | generic_declaration | generic_instantiation

10.1.1:
library_unit_renaming_declaration ::=
  package_renaming_declaration
| generic_renaming_declaration
| subprogram_renaming_declaration

10.1.1:
library_unit_body ::= subprogram_body | package_body

10.1.1:
parent_unit_name ::= name

10.1.2:
context_clause ::= {context_item}

10.1.2:
context_item ::= with_clause | use_clause

10.1.2:
with_clause ::= limited_with_clause | nonlimited_with_clause

10.1.2:
limited_with_clause ::= **limited** [**private**] **with** *library_unit*_name {, *library_unit*_name};

10.1.2:
nonlimited_with_clause ::= [**private**] **with** *library_unit*_name {, *library_unit*_name};

10.1.3:
body_stub ::=
   subprogram_body_stub | package_body_stub
 | task_body_stub | protected_body_stub

10.1.3:
subprogram_body_stub ::=
   [overriding_indicator]
   subprogram_specification **is separate**
     [aspect_specification];

10.1.3:
package_body_stub ::=
  package body defining_identifier **is separate**
    [aspect_specification];

10.1.3:
task_body_stub ::=
  task body defining_identifier **is separate**
    [aspect_specification];

10.1.3:
protected_body_stub ::=
  protected body defining_identifier **is separate**
    [aspect_specification];

10.1.3:
subunit ::= **separate** (parent_unit_name) proper_body

11.1:
exception_declaration ::= defining_identifier_list : **exception**
  [aspect_specification];

11.2:
handled_sequence_of_statements ::=
    sequence_of_statements
 [**exception**
    exception_handler

{exception_handler}]

11.2:
exception_handler ::=
when [choice_parameter_specification:] exception_choice {'|' exception_choice} =>
sequence_of_statements

11.2:
choice_parameter_specification ::= defining_identifier

11.2:
exception_choice ::= *exception_*name | **others**

11.3:
raise_statement ::= **raise**;
| **raise** *exception_*name [**with** *string_*expression];

11.3:
raise_expression ::= **raise** *exception_*name [**with** *string_*simple_expression]

12.1:
generic_declaration ::=
generic_subprogram_declaration | generic_package_declaration

12.1:
generic_subprogram_declaration ::=
generic_formal_part subprogram_specification
[aspect_specification];

12.1:
generic_package_declaration ::=
generic_formal_part package_specification;

12.1:
generic_formal_part ::=
generic {generic_formal_parameter_declaration | use_clause}

12.1:
generic_formal_parameter_declaration ::=
formal_object_declaration
| formal_type_declaration
| formal_subprogram_declaration
| formal_package_declaration

12.3:
generic_instantiation ::=
package defining_program_unit_name **is**
new *generic_package_*name [generic_actual_part]
[aspect_specification];
| [overriding_indicator]
procedure defining_program_unit_name **is**
new *generic_procedure_*name [generic_actual_part]
[aspect_specification];
| [overriding_indicator]
function defining_designator **is**
new *generic_function_*name [generic_actual_part]
[aspect_specification];

12.3:
generic_actual_part ::=
(generic_association {, generic_association})

12.3:
generic_association ::=
[*generic_formal_parameter_*selector_name =>] explicit_generic_actual_parameter

12.3:
explicit_generic_actual_parameter ::= expression | *variable_*name
| *subprogram_*name | *entry_*name | subtype_mark
| *package_instance_*name

12.4:

formal_object_declaration ::=
   defining_identifier_list : mode [null_exclusion] subtype_mark [:= default_expression]
      [aspect_specification];
 | defining_identifier_list : mode access_definition [:= default_expression]
      [aspect_specification];

12.5:
formal_type_declaration ::=
    formal_complete_type_declaration
  | formal_incomplete_type_declaration

12.5:
formal_complete_type_declaration ::=
   type defining_identifier[discriminant_part] **is** formal_type_definition
      [**or use** *default_*subtype_mark] [aspect_specification];

12.5:
formal_incomplete_type_declaration ::=
   type defining_identifier[discriminant_part] [**is tagged**]
      [**or use** *default_*subtype_mark];

12.5:
formal_type_definition ::=
    formal_private_type_definition
  | formal_derived_type_definition
  | formal_discrete_type_definition
  | formal_signed_integer_type_definition
  | formal_modular_type_definition
  | formal_floating_point_definition
  | formal_ordinary_fixed_point_definition
  | formal_decimal_fixed_point_definition
  | formal_array_type_definition
  | formal_access_type_definition
  | formal_interface_type_definition

12.5.1:
formal_private_type_definition ::= [[**abstract**] **tagged**] [**limited**] **private**

12.5.1:
formal_derived_type_definition ::=
   [**abstract**] [**limited** | **synchronized**] **new** subtype_mark [[**and** interface_list] **with private**]

12.5.2:
formal_discrete_type_definition ::= (<>)

12.5.2:
formal_signed_integer_type_definition ::= **range** <>

12.5.2:
formal_modular_type_definition ::= **mod** <>

12.5.2:
formal_floating_point_definition ::= **digits** <>

12.5.2:
formal_ordinary_fixed_point_definition ::= **delta** <>

12.5.2:
formal_decimal_fixed_point_definition ::= **delta** <> **digits** <>

12.5.3:
formal_array_type_definition ::= array_type_definition

12.5.4:
formal_access_type_definition ::= access_type_definition

12.5.5:
formal_interface_type_definition ::= interface_type_definition

12.6:
formal_subprogram_declaration ::= formal_concrete_subprogram_declaration
  | formal_abstract_subprogram_declaration

12.6:

```
formal_concrete_subprogram_declaration ::=
    with subprogram_specification [is subprogram_default]
      [aspect_specification];
```

12.6:

```
formal_abstract_subprogram_declaration ::=
    with subprogram_specification is abstract [subprogram_default]
      [aspect_specification];
```

12.6:

```
subprogram_default ::= default_name | <> | null
```

12.6:

```
default_name ::= name
```

12.7:

```
formal_package_declaration ::=
   with package defining_identifier is new generic_package_name formal_package_actual_part
      [aspect_specification];
```

12.7:

```
formal_package_actual_part ::=
   ([others =>] <>)
 | [generic_actual_part]
 | (formal_package_association {, formal_package_association} [, others => <>])
```

12.7:

```
formal_package_association ::=
   generic_association
 | generic_formal_parameter_selector_name => <>
```

13.1:

```
aspect_clause ::= attribute_definition_clause
     | enumeration_representation_clause
     | record_representation_clause
     | at_clause
```

13.1:

```
local_name ::= direct_name
     | direct_name'attribute_designator
     | library_unit_name
```

13.1.1:

```
aspect_specification ::=
  with aspect_mark [=> aspect_definition] {,
        aspect_mark [=> aspect_definition] }
```

13.1.1:

```
aspect_mark ::= aspect_identifier['Class]
```

13.1.1:

```
aspect_definition ::=
   name | expression | identifier
 | aggregate | global_aspect_definition
```

13.3:

```
attribute_definition_clause ::=
     for local_name'attribute_designator use expression;
   | for local_name'attribute_designator use name;
```

13.4:

```
enumeration_representation_clause ::=
   for first_subtype_local_name use enumeration_aggregate;
```

13.4:

```
enumeration_aggregate ::= array_aggregate
```

13.5.1:

```
record_representation_clause ::=
   for first_subtype_local_name use
     record [mod_clause]
       {component_clause}
```

end record [local_name];

13.5.1:
component_clause ::=
*component*_local_name **at** position **range** first_bit .. last_bit;

13.5.1:
position ::= *static*_expression

13.5.1:
first_bit ::= *static*_simple_expression

13.5.1:
last_bit ::= *static*_simple_expression

13.8:
code_statement ::= qualified_expression;

13.11.3:
storage_pool_indicator ::= *storage_pool*_name | **null** | Standard

13.12:
restriction ::= *restriction*_identifier
| *restriction_parameter*_identifier => restriction_parameter_argument

13.12:
restriction_parameter_argument ::= name | expression

H.7:
extended_global_mode ::=
overriding basic_global_mode

H.7.1:
formal_parameter_set ::=
formal_group_designator
| formal_parameter_name
| (formal_parameter_name{, formal_parameter_name})

H.7.1:
formal_group_designator ::= **null** | **all**

H.7.1:
formal_parameter_name ::=
*formal*_subtype_mark
| *formal_subprogram*_name
| *formal_access_to_subprogram_object*_name

H.7.1:
dispatching_operation_set ::=
dispatching_operation_specifier
| (dispatching_operation_specifier{, dispatching_operation_specifier})

H.7.1:
dispatching_operation_specifier ::=
*dispatching_operation*_name (*object*_name)

J.3:
delta_constraint ::= **delta** *static*_simple_expression [range_constraint]

J.7:
at_clause ::= **for** direct_name **use at** expression;

J.8:
mod_clause ::= **at mod** *static*_expression;

P.2 Syntax Cross Reference

{*AI05-0299-1*} In the following syntax cross reference, each syntactic category is followed by the subclause number where it is defined. In addition, each syntactic category *S* is followed by a list of the categories that use *S* in their definitions. For example, the first listing below shows that abort_statement appears in the definition of simple_statement. 1/3

abort_statement 9.8
  simple_statement 5.1

abortable_part 9.7.4
  asynchronous_select 9.7.4

abstract_subprogram_declaration 3.9.3
  basic_declaration 3.1

accept_alternative 9.7.1
  select_alternative 9.7.1

accept_statement 9.5.2
  accept_alternative 9.7.1
  compound_statement 5.1

access_definition 3.10
  component_definition 3.6
  discriminant_specification 3.7
  formal_object_declaration 12.4
  loop_parameter_subtype_indication 5.5.2
  object_declaration 3.3.1
  object_renaming_declaration 8.5.1
  parameter_and_result_profile 6.1
  parameter_specification 6.1
  return_subtype_indication 6.5

access_to_object_definition 3.10
  access_type_definition 3.10

access_to_subprogram_definition 3.10
  access_type_definition 3.10

access_type_definition 3.10
  formal_access_type_definition 12.5.4
  type_definition 3.2.1

actual_parameter_part 6.4
  entry_call_statement 9.5.3
  function_call 6.4
  generalized_indexing 4.1.6
  procedure_call_statement 6.4

aggregate 4.3
  aspect_definition 13.1.1
  expression_function_declaration 6.8
  primary 4.4
  qualified_expression 4.7

allocator 4.8
  primary 4.4

ancestor_part 4.3.2
  extension_aggregate 4.3.2

array_aggregate 4.3.3
  aggregate 4.3
  enumeration_aggregate 13.4

array_component_association 4.3.3
  array_component_association_list 4.3.3

array_component_association_list 4.3.3
  array_delta_aggregate 4.3.4
  named_array_aggregate 4.3.3

array_delta_aggregate 4.3.4
  delta_aggregate 4.3.4

array_type_definition 3.6
  formal_array_type_definition 12.5.3
  object_declaration 3.3.1
  type_definition 3.2.1

aspect_clause 13.1
  basic_declarative_item 3.11
  component_item 3.8
  protected_operation_declaration 9.4
  protected_operation_item 9.4
  task_item 9.1

aspect_definition 13.1.1
  aspect_specification 13.1.1

aspect_mark 13.1.1
  aspect_specification 13.1.1
  pragma_argument_association 2.8

aspect_specification 13.1.1
  abstract_subprogram_declaration 3.9.3
  component_declaration 3.8
  discriminant_specification 3.7
  entry_body 9.5.2
  entry_declaration 9.5.2
  entry_index_specification 9.5.2
  exception_declaration 11.1
  exception_renaming_declaration 8.5.2
  expression_function_declaration 6.8
  extended_return_object_declaration 6.5
  formal_abstract_subprogram_declaration 12.6
  formal_complete_type_declaration 12.5
  formal_concrete_subprogram_declaration 12.6
  formal_object_declaration 12.4
  formal_package_declaration 12.7
  full_type_declaration 3.2.1
  generic_instantiation 12.3
  generic_renaming_declaration 8.5.5
  generic_subprogram_declaration 12.1
  iteration_scheme 5.5
  null_procedure_declaration 6.7
  object_declaration 3.3.1
  object_renaming_declaration 8.5.1
  package_body 7.2
  package_body_stub 10.1.3
  package_renaming_declaration 8.5.3
  package_specification 7.1
  parallel_block_statement 5.6.1
  parameter_specification 6.1
  private_extension_declaration 7.3
  private_type_declaration 7.3
  protected_body 9.4
  protected_body_stub 10.1.3
  protected_type_declaration 9.4
  single_protected_declaration 9.4
  single_task_declaration 9.1
  subprogram_body 6.3
  subprogram_body_stub 10.1.3
  subprogram_declaration 6.1
  subprogram_renaming_declaration 8.5.4
  subtype_declaration 3.2.2
  task_body 9.1
  task_body_stub 10.1.3
  task_type_declaration 9.1
  value_sequence 4.5.10

assignment_statement 5.2

simple_statement 5.1

asynchronous_select 9.7.4
select_statement 9.7

at_clause J.7
aspect_clause 13.1

attribute_definition_clause 13.3
aspect_clause 13.1

attribute_designator 4.1.4
attribute_definition_clause 13.3
attribute_reference 4.1.4
local_name 13.1

attribute_reference 4.1.4
name 4.1

base 2.4.2
based_literal 2.4.2

based_literal 2.4.2
numeric_literal 2.4

based_numeral 2.4.2
based_literal 2.4.2

basic_declaration 3.1
basic_declarative_item 3.11

basic_declarative_item 3.11
declarative_item 3.11
package_specification 7.1

basic_global_mode 6.1.2
extended_global_mode H.7
global_mode 6.1.2

binary_adding_operator 4.5
simple_expression 4.4

block_statement 5.6
compound_statement 5.1

body 3.11
declarative_item 3.11

body_stub 10.1.3
body 3.11

case_expression 4.5.7
conditional_expression 4.5.7

case_expression_alternative 4.5.7
case_expression 4.5.7

case_statement 5.4
compound_statement 5.1

case_statement_alternative 5.4
case_statement 5.4

character 2.1
comment 2.7

character_literal 2.5
defining_character_literal 3.5.1
name 4.1
selector_name 4.1.3

choice_expression 4.4
discrete_choice 3.8.1

choice_parameter_specification 11.2
exception_handler 11.2

choice_relation 4.4
choice_expression 4.4

chunk_specification 5.5
iteration_scheme 5.5
parallel_block_statement 5.6.1
value_sequence 4.5.10

code_statement 13.8
simple_statement 5.1

compilation_unit 10.1.1
compilation 10.1.1

component_choice_list 4.3.1
record_component_association 4.3.1

component_clause 13.5.1
record_representation_clause 13.5.1

component_declaration 3.8
component_item 3.8
protected_element_declaration 9.4

component_definition 3.6
component_declaration 3.8
constrained_array_definition 3.6
unconstrained_array_definition 3.6

component_item 3.8
component_list 3.8

component_list 3.8
record_definition 3.8
variant 3.8.1

composite_constraint 3.2.2
constraint 3.2.2

compound_statement 5.1
statement 5.1

condition 4.5.7
entry_barrier 9.5.2
exit_statement 5.7
guard 9.7.1
if_expression 4.5.7
if_statement 5.3
iteration_scheme 5.5
iterator_filter 5.5

conditional_entry_call 9.7.3
select_statement 9.7

conditional_expression 4.5.7
primary 4.4

constrained_array_definition 3.6
array_type_definition 3.6

constraint 3.2.2
subtype_indication 3.2.2

container_aggregate 4.3.5

aggregate 4.3

container_element_association 4.3.5
container_element_association_list 4.3.5

container_element_association_list 4.3.5
named_container_aggregate 4.3.5

context_clause 10.1.2
compilation_unit 10.1.1

context_item 10.1.2
context_clause 10.1.2

decimal_fixed_point_definition 3.5.9
fixed_point_definition 3.5.9

decimal_literal 2.4.1
numeric_literal 2.4

declarative_item 3.11
declarative_part 3.11

declarative_part 3.11
block_statement 5.6
entry_body 9.5.2
package_body 7.2
subprogram_body 6.3
task_body 9.1

declare_expression 4.5.9
primary 4.4

declare_item 4.5.9
declare_expression 4.5.9

default_expression 3.7
component_declaration 3.8
discriminant_specification 3.7
formal_object_declaration 12.4
parameter_specification 6.1

default_name 12.6
subprogram_default 12.6

defining_character_literal 3.5.1
enumeration_literal_specification 3.5.1

defining_designator 6.1
function_specification 6.1
generic_instantiation 12.3

defining_identifier 3.1
choice_parameter_specification 11.2
chunk_specification 5.5
defining_identifier_list 3.3.1
defining_program_unit_name 6.1
entry_body 9.5.2
entry_declaration 9.5.2
entry_index_specification 9.5.2
enumeration_literal_specification 3.5.1
exception_renaming_declaration 8.5.2
extended_return_object_declaration 6.5
formal_complete_type_declaration 12.5
formal_incomplete_type_declaration 12.5
formal_package_declaration 12.7
full_type_declaration 3.2.1
incomplete_type_declaration 3.10.1
iterated_component_association 4.3.3
iterator_parameter_specification 5.5.3
iterator_specification 5.5.2
loop_parameter_specification 5.5
object_renaming_declaration 8.5.1
package_body_stub 10.1.3
private_extension_declaration 7.3
private_type_declaration 7.3
protected_body 9.4
protected_body_stub 10.1.3
protected_type_declaration 9.4
single_protected_declaration 9.4
single_task_declaration 9.1
subtype_declaration 3.2.2
task_body 9.1
task_body_stub 10.1.3
task_type_declaration 9.1

defining_identifier_list 3.3.1
component_declaration 3.8
discriminant_specification 3.7
exception_declaration 11.1
formal_object_declaration 12.4
number_declaration 3.3.2
object_declaration 3.3.1
parameter_specification 6.1

defining_operator_symbol 6.1
defining_designator 6.1

defining_program_unit_name 6.1
defining_designator 6.1
generic_instantiation 12.3
generic_renaming_declaration 8.5.5
package_body 7.2
package_renaming_declaration 8.5.3
package_specification 7.1
procedure_specification 6.1

delay_alternative 9.7.1
select_alternative 9.7.1
timed_entry_call 9.7.2

delay_relative_statement 9.6
delay_statement 9.6

delay_statement 9.6
delay_alternative 9.7.1
simple_statement 5.1
triggering_statement 9.7.4

delay_until_statement 9.6
delay_statement 9.6

delta_aggregate 4.3.4
aggregate 4.3

delta_constraint J.3
scalar_constraint 3.2.2

derived_type_definition 3.4
type_definition 3.2.1

designator 6.1
subprogram_body 6.3

digit 2.4.1
extended_digit 2.4.2
numeral 2.4.1

digits_constraint 3.5.9
scalar_constraint 3.2.2

direct_name 4.1
accept_statement 9.5.2
at_clause J.7
local_name 13.1
name 4.1
statement_identifier 5.1
variant_part 3.8.1

discrete_choice 3.8.1
discrete_choice_list 3.8.1

discrete_choice_list 3.8.1
array_component_association 4.3.3
case_expression_alternative 4.5.7
case_statement_alternative 5.4
iterated_component_association 4.3.3
variant 3.8.1

discrete_range 3.6.1
index_constraint 3.6.1
key_choice 4.3.5
slice 4.1.2

discrete_subtype_definition 3.6
chunk_specification 5.5
constrained_array_definition 3.6
entry_declaration 9.5.2
entry_index_specification 9.5.2
loop_parameter_specification 5.5

discriminant_association 3.7.1
discriminant_constraint 3.7.1

discriminant_constraint 3.7.1
composite_constraint 3.2.2

discriminant_part 3.7
formal_complete_type_declaration 12.5
formal_incomplete_type_declaration 12.5
incomplete_type_declaration 3.10.1
private_extension_declaration 7.3
private_type_declaration 7.3

discriminant_specification 3.7
known_discriminant_part 3.7

dispatching_operation_specifier H.7.1
dispatching_operation_set H.7.1

entry_barrier 9.5.2
entry_body 9.5.2

entry_body 9.5.2
protected_operation_item 9.4

entry_body_formal_part 9.5.2
entry_body 9.5.2

entry_call_alternative 9.7.2
conditional_entry_call 9.7.3
timed_entry_call 9.7.2

entry_call_statement 9.5.3
procedure_or_entry_call 9.7.2
simple_statement 5.1

entry_declaration 9.5.2
protected_operation_declaration 9.4
task_item 9.1

entry_index 9.5.2
accept_statement 9.5.2

entry_index_specification 9.5.2
entry_body_formal_part 9.5.2

enumeration_aggregate 13.4
enumeration_representation_clause 13.4

enumeration_literal_specification 3.5.1
enumeration_type_definition 3.5.1

enumeration_representation_clause 13.4
aspect_clause 13.1

enumeration_type_definition 3.5.1
type_definition 3.2.1

exception_choice 11.2
exception_handler 11.2

exception_declaration 11.1
basic_declaration 3.1

exception_handler 11.2
handled_sequence_of_statements 11.2

exception_renaming_declaration 8.5.2
renaming_declaration 8.5

exit_statement 5.7
simple_statement 5.1

explicit_actual_parameter 6.4
parameter_association 6.4

explicit_dereference 4.1
name 4.1

explicit_generic_actual_parameter 12.3
generic_association 12.3

exponent 2.4.1
based_literal 2.4.2
decimal_literal 2.4.1

expression 4.4
ancestor_part 4.3.2
array_component_association 4.3.3
array_delta_aggregate 4.3.4
aspect_definition 13.1.1
assignment_statement 5.2
at_clause J.7
attribute_definition_clause 13.3
attribute_designator 4.1.4
case_expression 4.5.7
case_expression_alternative 4.5.7
case_statement 5.4
condition 4.5.7
container_element_association 4.3.5
decimal_fixed_point_definition 3.5.9
declare_expression 4.5.9
default_expression 3.7
delay_relative_statement 9.6
delay_until_statement 9.6

discriminant_association 3.7.1
entry_index 9.5.2
explicit_actual_parameter 6.4
explicit_generic_actual_parameter 12.3
expression_function_declaration 6.8
extended_return_object_declaration 6.5
floating_point_definition 3.5.7
if_expression 4.5.7
indexed_component 4.1.1
iterated_component_association 4.3.3
iterated_element_association 4.3.5
key_choice 4.3.5
mod_clause J.8
modular_type_definition 3.5.4
number_declaration 3.3.2
object_declaration 3.3.1
ordinary_fixed_point_definition 3.5.9
position 13.5.1
positional_array_aggregate 4.3.3
positional_container_aggregate 4.3.5
pragma_argument_association 2.8
predicate 4.5.8
primary 4.4
qualified_expression 4.7
raise_statement 11.3
range_attribute_designator 4.1.4
record_component_association 4.3.1
record_delta_aggregate 4.3.4
reduction_specification 4.5.10
restriction_parameter_argument 13.12
simple_return_statement 6.5
type_conversion 4.6

expression_function_declaration 6.8
basic_declaration 3.1
protected_operation_item 9.4

extended_digit 2.4.2
based_numeral 2.4.2

extended_global_mode H.7
global_mode 6.1.2

extended_return_object_declaration 6.5
extended_return_statement 6.5

extended_return_statement 6.5
compound_statement 5.1

extension_aggregate 4.3.2
aggregate 4.3

factor 4.4
term 4.4

first_bit 13.5.1
component_clause 13.5.1

fixed_point_definition 3.5.9
real_type_definition 3.5.6

floating_point_definition 3.5.7
real_type_definition 3.5.6

formal_abstract_subprogram_declaration 12.6
formal_subprogram_declaration 12.6

formal_access_type_definition 12.5.4
formal_type_definition 12.5

formal_array_type_definition 12.5.3
formal_type_definition 12.5

formal_complete_type_declaration 12.5
formal_type_declaration 12.5

formal_concrete_subprogram_declaration 12.6
formal_subprogram_declaration 12.6

formal_decimal_fixed_point_definition 12.5.2
formal_type_definition 12.5

formal_derived_type_definition 12.5.1
formal_type_definition 12.5

formal_discrete_type_definition 12.5.2
formal_type_definition 12.5

formal_floating_point_definition 12.5.2
formal_type_definition 12.5

formal_group_designator H.7.1
formal_parameter_set H.7.1

formal_incomplete_type_declaration 12.5
formal_type_declaration 12.5

formal_interface_type_definition 12.5.5
formal_type_definition 12.5

formal_modular_type_definition 12.5.2
formal_type_definition 12.5

formal_object_declaration 12.4
generic_formal_parameter_declaration 12.1

formal_ordinary_fixed_point_definition 12.5.2
formal_type_definition 12.5

formal_package_actual_part 12.7
formal_package_declaration 12.7

formal_package_association 12.7
formal_package_actual_part 12.7

formal_package_declaration 12.7
generic_formal_parameter_declaration 12.1

formal_parameter_name H.7.1
formal_parameter_set H.7.1

formal_part 6.1
iterator_parameter_specification 5.5.3
parameter_and_result_profile 6.1
parameter_profile 6.1

formal_private_type_definition 12.5.1
formal_type_definition 12.5

formal_signed_integer_type_definition 12.5.2
formal_type_definition 12.5

formal_subprogram_declaration 12.6
generic_formal_parameter_declaration 12.1

formal_type_declaration 12.5
generic_formal_parameter_declaration 12.1

formal_type_definition 12.5
formal_complete_type_declaration 12.5

full_type_declaration 3.2.1
type_declaration 3.2.1

function_call 6.4
name 4.1

function_specification 6.1
expression_function_declaration 6.8
subprogram_specification 6.1

general_access_modifier 3.10
access_to_object_definition 3.10

generalized_indexing 4.1.6
name 4.1

generalized_reference 4.1.5
name 4.1

generic_actual_part 12.3
formal_package_actual_part 12.7
generic_instantiation 12.3

generic_association 12.3
formal_package_association 12.7
generic_actual_part 12.3

generic_declaration 12.1
basic_declaration 3.1
library_unit_declaration 10.1.1

generic_formal_parameter_declaration 12.1
generic_formal_part 12.1

generic_formal_part 12.1
generic_package_declaration 12.1
generic_subprogram_declaration 12.1

generic_instantiation 12.3
basic_declaration 3.1
library_unit_declaration 10.1.1

generic_package_declaration 12.1
generic_declaration 12.1

generic_renaming_declaration 8.5.5
library_unit_renaming_declaration 10.1.1
renaming_declaration 8.5

generic_subprogram_declaration 12.1
generic_declaration 12.1

global_aspect_definition 6.1.2
aspect_definition 13.1.1

global_aspect_element 6.1.2
global_aspect_definition 6.1.2

global_designator 6.1.2
global_aspect_definition 6.1.2

global_mode 6.1.2
global_aspect_definition 6.1.2
global_aspect_element 6.1.2

global_name 6.1.2
global_designator 6.1.2
global_set 6.1.2

global_set 6.1.2
global_aspect_element 6.1.2

goto_statement 5.8
simple_statement 5.1

graphic_character 2.1
character_literal 2.5
string_element 2.6

guard 9.7.1
selective_accept 9.7.1

handled_sequence_of_statements 11.2
accept_statement 9.5.2
block_statement 5.6
entry_body 9.5.2
extended_return_statement 6.5
package_body 7.2
subprogram_body 6.3
task_body 9.1

identifier 2.3
accept_statement 9.5.2
aspect_definition 13.1.1
aspect_mark 13.1.1
attribute_designator 4.1.4
block_statement 5.6
defining_identifier 3.1
designator 6.1
direct_name 4.1
entry_body 9.5.2
loop_statement 5.5
package_body 7.2
package_specification 7.1
pragma 2.8
pragma_argument_association 2.8
protected_body 9.4
protected_definition 9.4
record_definition 3.8
reduction_attribute_designator 4.5.10
restriction 13.12
selector_name 4.1.3
task_body 9.1
task_definition 9.1

identifier_extend 2.3
identifier 2.3

identifier_start 2.3
identifier 2.3

if_expression 4.5.7
conditional_expression 4.5.7

if_statement 5.3
compound_statement 5.1

implicit_dereference 4.1
prefix 4.1

incomplete_type_declaration 3.10.1
type_declaration 3.2.1

index_constraint 3.6.1
composite_constraint 3.2.2

index_subtype_definition 3.6
unconstrained_array_definition 3.6

indexed_component 4.1.1

name 4.1

integer_type_definition 3.5.4
type_definition 3.2.1

interface_list 3.9.4
derived_type_definition 3.4
formal_derived_type_definition 12.5.1
interface_type_definition 3.9.4
private_extension_declaration 7.3
protected_type_declaration 9.4
single_protected_declaration 9.4
single_task_declaration 9.1
task_type_declaration 9.1

interface_type_definition 3.9.4
formal_interface_type_definition 12.5.5
type_definition 3.2.1

iterated_component_association 4.3.3
array_component_association 4.3.3

iterated_element_association 4.3.5
container_element_association 4.3.5
value_sequence 4.5.10

iteration_scheme 5.5
loop_statement 5.5

iterator_actual_parameter_part 5.5.3
iterator_procedure_call 5.5.3

iterator_filter 5.5
iterator_specification 5.5.2
loop_parameter_specification 5.5
procedural_iterator 5.5.3

iterator_parameter_association 5.5.3
iterator_actual_parameter_part 5.5.3

iterator_parameter_specification 5.5.3
procedural_iterator 5.5.3

iterator_procedure_call 5.5.3
procedural_iterator 5.5.3

iterator_specification 5.5.2
iterated_component_association 4.3.3
iterated_element_association 4.3.5
iteration_scheme 5.5
quantified_expression 4.5.8

key_choice 4.3.5
key_choice_list 4.3.5

key_choice_list 4.3.5
container_element_association 4.3.5

known_discriminant_part 3.7
discriminant_part 3.7
full_type_declaration 3.2.1
protected_type_declaration 9.4
task_type_declaration 9.1

label 5.1
sequence_of_statements 5.1
statement 5.1

last_bit 13.5.1
component_clause 13.5.1

letter_lowercase ...
identifier_start 2.3

letter_modifier ...
identifier_start 2.3

letter_other ...
identifier_start 2.3

letter_titlecase ...
identifier_start 2.3

letter_uppercase ...
identifier_start 2.3

library_item 10.1.1
compilation_unit 10.1.1

library_unit_body 10.1.1
library_item 10.1.1

library_unit_declaration 10.1.1
library_item 10.1.1

library_unit_renaming_declaration 10.1.1
library_item 10.1.1

limited_with_clause 10.1.2
with_clause 10.1.2

local_name 13.1
attribute_definition_clause 13.3
component_clause 13.5.1
enumeration_representation_clause 13.4
record_representation_clause 13.5.1

loop_parameter_specification 5.5
iterated_element_association 4.3.5
iteration_scheme 5.5
quantified_expression 4.5.8

loop_parameter_subtype_indication 5.5.2
iterator_specification 5.5.2

loop_statement 5.5
compound_statement 5.1

mark_non_spacing ...
identifier_extend 2.3

mark_spacing_combining ...
identifier_extend 2.3

membership_choice 4.4
membership_choice_list 4.4

membership_choice_list 4.4
relation 4.4

mod_clause J.8
record_representation_clause 13.5.1

mode 6.1
formal_object_declaration 12.4
parameter_specification 6.1

modular_type_definition 3.5.4
integer_type_definition 3.5.4

multiplying_operator 4.5
term 4.4

name 4.1
- abort_statement 9.8
- aspect_definition 13.1.1
- assignment_statement 5.2
- attribute_definition_clause 13.3
- default_name 12.6
- dispatching_operation_specifier H.7.1
- entry_call_statement 9.5.3
- exception_choice 11.2
- exception_renaming_declaration 8.5.2
- exit_statement 5.7
- explicit_actual_parameter 6.4
- explicit_dereference 4.1
- explicit_generic_actual_parameter 12.3
- formal_package_declaration 12.7
- formal_parameter_name H.7.1
- function_call 6.4
- generalized_reference 4.1.5
- generic_instantiation 12.3
- generic_renaming_declaration 8.5.5
- global_name 6.1.2
- goto_statement 5.8
- implicit_dereference 4.1
- iterator_procedure_call 5.5.3
- iterator_specification 5.5.2
- limited_with_clause 10.1.2
- local_name 13.1
- nonlimited_with_clause 10.1.2
- object_renaming_declaration 8.5.1
- package_renaming_declaration 8.5.3
- parent_unit_name 10.1.1
- pragma_argument_association 2.8
- prefix 4.1
- primary 4.4
- procedure_call_statement 6.4
- raise_expression 11.3
- raise_statement 11.3
- reduction_specification 4.5.10
- requeue_statement 9.5.4
- restriction_parameter_argument 13.12
- storage_pool_indicator 13.11.3
- subpool_specification 4.8
- subprogram_renaming_declaration 8.5.4
- subtype_mark 3.2.2
- type_conversion 4.6
- use_package_clause 8.4

named_array_aggregate 4.3.3
- array_aggregate 4.3.3

named_container_aggregate 4.3.5
- container_aggregate 4.3.5

nonlimited_with_clause 10.1.2
- with_clause 10.1.2

null_array_aggregate 4.3.3
- array_aggregate 4.3.3

null_container_aggregate 4.3.5
- container_aggregate 4.3.5

null_exclusion 3.10
- access_definition 3.10
- access_type_definition 3.10
- discriminant_specification 3.7
- formal_object_declaration 12.4
- object_renaming_declaration 8.5.1
- parameter_and_result_profile 6.1
- parameter_specification 6.1
- subtype_indication 3.2.2

null_procedure_declaration 6.7
- basic_declaration 3.1
- protected_operation_item 9.4

null_statement 5.1
- simple_statement 5.1

number_decimal ...
- identifier_extend 2.3

number_declaration 3.3.2
- basic_declaration 3.1

number_letter ...
- identifier_start 2.3

numeral 2.4.1
- base 2.4.2
- decimal_literal 2.4.1
- exponent 2.4.1

numeric_literal 2.4
- primary 4.4

object_declaration 3.3.1
- basic_declaration 3.1
- declare_item 4.5.9

object_renaming_declaration 8.5.1
- declare_item 4.5.9
- renaming_declaration 8.5

operator_symbol 6.1
- defining_operator_symbol 6.1
- designator 6.1
- direct_name 4.1
- selector_name 4.1.3

ordinary_fixed_point_definition 3.5.9
- fixed_point_definition 3.5.9

overriding_indicator 8.3.1
- abstract_subprogram_declaration 3.9.3
- entry_declaration 9.5.2
- expression_function_declaration 6.8
- generic_instantiation 12.3
- null_procedure_declaration 6.7
- subprogram_body 6.3
- subprogram_body_stub 10.1.3
- subprogram_declaration 6.1
- subprogram_renaming_declaration 8.5.4

package_body 7.2
- library_unit_body 10.1.1
- proper_body 3.11

package_body_stub 10.1.3
- body_stub 10.1.3

package_declaration 7.1
- basic_declaration 3.1
- library_unit_declaration 10.1.1

package_renaming_declaration 8.5.3
  library_unit_renaming_declaration 10.1.1
  renaming_declaration 8.5

package_specification 7.1
  generic_package_declaration 12.1
  package_declaration 7.1

parallel_block_statement 5.6.1
  compound_statement 5.1

parameter_and_result_profile 6.1
  access_definition 3.10
  access_to_subprogram_definition 3.10
  function_specification 6.1

parameter_association 6.4
  actual_parameter_part 6.4
  iterator_parameter_association 5.5.3

parameter_association_with_box 5.5.3
  iterator_parameter_association 5.5.3

parameter_profile 6.1
  accept_statement 9.5.2
  access_definition 3.10
  access_to_subprogram_definition 3.10
  entry_body_formal_part 9.5.2
  entry_declaration 9.5.2
  procedure_specification 6.1

parameter_specification 6.1
  formal_part 6.1

parent_unit_name 10.1.1
  defining_program_unit_name 6.1
  designator 6.1
  package_body 7.2
  package_specification 7.1
  subunit 10.1.3

position 13.5.1
  component_clause 13.5.1

positional_array_aggregate 4.3.3
  array_aggregate 4.3.3

positional_container_aggregate 4.3.5
  container_aggregate 4.3.5

pragma_argument_association 2.8
  pragma 2.8

predicate 4.5.8
  quantified_expression 4.5.8

prefix 4.1
  attribute_reference 4.1.4
  function_call 6.4
  generalized_indexing 4.1.6
  indexed_component 4.1.1
  iterator_procedure_call 5.5.3
  procedure_call_statement 6.4
  range_attribute_reference 4.1.4
  reduction_attribute_reference 4.5.10
  selected_component 4.1.3
  slice 4.1.2

primary 4.4
  factor 4.4

private_extension_declaration 7.3
  type_declaration 3.2.1

private_type_declaration 7.3
  type_declaration 3.2.1

procedural_iterator 5.5.3
  iteration_scheme 5.5

procedure_call_statement 6.4
  procedure_or_entry_call 9.7.2
  simple_statement 5.1

procedure_or_entry_call 9.7.2
  entry_call_alternative 9.7.2
  triggering_statement 9.7.4

procedure_specification 6.1
  null_procedure_declaration 6.7
  subprogram_specification 6.1

proper_body 3.11
  body 3.11
  subunit 10.1.3

protected_body 9.4
  proper_body 3.11

protected_body_stub 10.1.3
  body_stub 10.1.3

protected_definition 9.4
  protected_type_declaration 9.4
  single_protected_declaration 9.4

protected_element_declaration 9.4
  protected_definition 9.4

protected_operation_declaration 9.4
  protected_definition 9.4
  protected_element_declaration 9.4

protected_operation_item 9.4
  protected_body 9.4

protected_type_declaration 9.4
  full_type_declaration 3.2.1

punctuation_connector ...
  identifier_extend 2.3

qualified_expression 4.7
  allocator 4.8
  code_statement 13.8
  name 4.1

quantified_expression 4.5.8
  primary 4.4

quantifier 4.5.8
  quantified_expression 4.5.8

raise_expression 11.3
  relation 4.4

raise_statement 11.3
  simple_statement 5.1

range 3.5

discrete_choice 3.8.1
discrete_range 3.6.1
discrete_subtype_definition 3.6
membership_choice 4.4
range_constraint 3.5

range_attribute_designator 4.1.4
range_attribute_reference 4.1.4

range_attribute_reference 4.1.4
range 3.5

range_constraint 3.5
delta_constraint J.3
digits_constraint 3.5.9
scalar_constraint 3.2.2

real_range_specification 3.5.7
decimal_fixed_point_definition 3.5.9
floating_point_definition 3.5.7
ordinary_fixed_point_definition 3.5.9

real_type_definition 3.5.6
type_definition 3.2.1

record_aggregate 4.3.1
aggregate 4.3

record_component_association 4.3.1
record_component_association_list 4.3.1

record_component_association_list 4.3.1
extension_aggregate 4.3.2
record_aggregate 4.3.1
record_delta_aggregate 4.3.4

record_definition 3.8
record_extension_part 3.9.1
record_type_definition 3.8

record_delta_aggregate 4.3.4
delta_aggregate 4.3.4

record_extension_part 3.9.1
derived_type_definition 3.4

record_representation_clause 13.5.1
aspect_clause 13.1

record_type_definition 3.8
type_definition 3.2.1

reduction_attribute_designator 4.5.10
reduction_attribute_reference 4.5.10

reduction_attribute_reference 4.5.10
attribute_reference 4.1.4

reduction_specification 4.5.10
reduction_attribute_designator 4.5.10

relation 4.4
expression 4.4

relational_operator 4.5
choice_relation 4.4
relation 4.4

renaming_declaration 8.5
basic_declaration 3.1

requeue_statement 9.5.4
simple_statement 5.1

restriction_parameter_argument 13.12
restriction 13.12

return_subtype_indication 6.5
extended_return_object_declaration 6.5

scalar_constraint 3.2.2
constraint 3.2.2

select_alternative 9.7.1
selective_accept 9.7.1

select_statement 9.7
compound_statement 5.1

selected_component 4.1.3
name 4.1

selective_accept 9.7.1
select_statement 9.7

selector_name 4.1.3
component_choice_list 4.3.1
discriminant_association 3.7.1
formal_package_association 12.7
generic_association 12.3
parameter_association 6.4
parameter_association_with_box 5.5.3
selected_component 4.1.3

sequence_of_statements 5.1
abortable_part 9.7.4
accept_alternative 9.7.1
case_statement_alternative 5.4
conditional_entry_call 9.7.3
delay_alternative 9.7.1
entry_call_alternative 9.7.2
exception_handler 11.2
handled_sequence_of_statements 11.2
if_statement 5.3
loop_statement 5.5
parallel_block_statement 5.6.1
selective_accept 9.7.1
triggering_alternative 9.7.4

signed_integer_type_definition 3.5.4
integer_type_definition 3.5.4

simple_expression 4.4
choice_relation 4.4
chunk_specification 5.5
delta_constraint J.3
digits_constraint 3.5.9
first_bit 13.5.1
last_bit 13.5.1
membership_choice 4.4
raise_expression 11.3
range 3.5
real_range_specification 3.5.7
relation 4.4
signed_integer_type_definition 3.5.4

simple_return_statement 6.5
simple_statement 5.1

simple_statement 5.1
  statement 5.1

single_protected_declaration 9.4
  object_declaration 3.3.1

single_task_declaration 9.1
  object_declaration 3.3.1

slice 4.1.2
  name 4.1

statement 5.1
  sequence_of_statements 5.1

statement_identifier 5.1
  block_statement 5.6
  label 5.1
  loop_statement 5.5

string_element 2.6
  string_literal 2.6

string_literal 2.6
  operator_symbol 6.1
  primary 4.4

subpool_specification 4.8
  allocator 4.8

subprogram_body 6.3
  library_unit_body 10.1.1
  proper_body 3.11
  protected_operation_item 9.4

subprogram_body_stub 10.1.3
  body_stub 10.1.3

subprogram_declaration 6.1
  basic_declaration 3.1
  library_unit_declaration 10.1.1
  protected_operation_declaration 9.4
  protected_operation_item 9.4

subprogram_default 12.6
  formal_abstract_subprogram_declaration 12.6
  formal_concrete_subprogram_declaration 12.6

subprogram_renaming_declaration 8.5.4
  library_unit_renaming_declaration 10.1.1
  renaming_declaration 8.5

subprogram_specification 6.1
  abstract_subprogram_declaration 3.9.3
  formal_abstract_subprogram_declaration 12.6
  formal_concrete_subprogram_declaration 12.6
  generic_subprogram_declaration 12.1
  subprogram_body 6.3
  subprogram_body_stub 10.1.3
  subprogram_declaration 6.1
  subprogram_renaming_declaration 8.5.4

subtype_declaration 3.2.2
  basic_declaration 3.1

subtype_indication 3.2.2
  access_to_object_definition 3.10
  allocator 4.8
  component_definition 3.6
  derived_type_definition 3.4
  discrete_choice 3.8.1
  discrete_range 3.6.1
  discrete_subtype_definition 3.6
  loop_parameter_subtype_indication 5.5.2
  object_declaration 3.3.1
  private_extension_declaration 7.3
  return_subtype_indication 6.5
  subtype_declaration 3.2.2

subtype_mark 3.2.2
  access_definition 3.10
  ancestor_part 4.3.2
  discriminant_specification 3.7
  explicit_generic_actual_parameter 12.3
  formal_complete_type_declaration 12.5
  formal_derived_type_definition 12.5.1
  formal_incomplete_type_declaration 12.5
  formal_object_declaration 12.4
  formal_parameter_name H.7.1
  index_subtype_definition 3.6
  interface_list 3.9.4
  membership_choice 4.4
  object_renaming_declaration 8.5.1
  parameter_and_result_profile 6.1
  parameter_specification 6.1
  qualified_expression 4.7
  subtype_indication 3.2.2
  type_conversion 4.6
  use_type_clause 8.4

subunit 10.1.3
  compilation_unit 10.1.1

target_name 5.2.1
  name 4.1

task_body 9.1
  proper_body 3.11

task_body_stub 10.1.3
  body_stub 10.1.3

task_definition 9.1
  single_task_declaration 9.1
  task_type_declaration 9.1

task_item 9.1
  task_definition 9.1

task_type_declaration 9.1
  full_type_declaration 3.2.1

term 4.4
  simple_expression 4.4

terminate_alternative 9.7.1
  select_alternative 9.7.1

timed_entry_call 9.7.2
  select_statement 9.7

triggering_alternative 9.7.4
  asynchronous_select 9.7.4

triggering_statement 9.7.4
  triggering_alternative 9.7.4

type_conversion 4.6

name 4.1

type_declaration 3.2.1
basic_declaration 3.1

type_definition 3.2.1
full_type_declaration 3.2.1

unary_adding_operator 4.5
simple_expression 4.4

unconstrained_array_definition 3.6
array_type_definition 3.6

underline ...
based_numeral 2.4.2
numeral 2.4.1

unknown_discriminant_part 3.7
discriminant_part 3.7

use_clause 8.4
basic_declarative_item 3.11
context_item 10.1.2
generic_formal_part 12.1

use_package_clause 8.4
use_clause 8.4

use_type_clause 8.4
use_clause 8.4

value_sequence 4.5.10
reduction_attribute_reference 4.5.10

variant 3.8.1
variant_part 3.8.1

variant_part 3.8.1
component_list 3.8

with_clause 10.1.2
context_item 10.1.2

Annex Q
(informative)
Language-Defined Entities

{*AI95-00440-01*} This annex lists the language-defined entities of the language. A list of language-defined library units can be found in Annex A, "Predefined Language Environment". 1/2

Q.1 Language-Defined Packages

{*AI95-00440-01*} {*AI05-0299-1*} This subclause lists all language-defined packages. 1/3

Ada A.2(2/5)
Address_To_Access_Conversions
  child of System 13.7.2(2/5)
Arithmetic
  child of Ada.Calendar 9.6.1(8/5)
ASCII
  in Standard A.1(36.3/2)
Assertions
  child of Ada 11.4.2(12/5)
Asynchronous_Task_Control
  child of Ada D.11(3/5)
Atomic_Operations
  child of System C.6.1(3/5)
Big_Integers
  child of Ada.Numerics.Big_Numbers A.5.5(3/5), A.5.6(2/5)
Big_Reals
  child of Ada.Numerics.Big_Numbers A.5.7(2/5)
Bounded
  child of Ada.Streams.Storage 13.13.1(26/5)
  child of Ada.Strings A.4.4(3/5)
  child of Ada.Strings.Text_Buffers A.4.12(26/5)
Bounded_Doubly_Linked_Lists
  child of Ada.Containers A.18.20(1/3)
Bounded_Hashed_Maps
  child of Ada.Containers A.18.21(1/3)
Bounded_Hashed_Sets
  child of Ada.Containers A.18.23(1/3)
Bounded_Indefinite_Holders
  child of Ada.Containers A.18.32(1/5)
Bounded_IO
  child of Ada.Text_IO A.10.11(3/5)
Bounded_Multiway_Trees
  child of Ada.Containers A.18.25(1/3)
Bounded_Ordered_Maps
  child of Ada.Containers A.18.22(1/3)
Bounded_Ordered_Sets
  child of Ada.Containers A.18.24(1/3)
Bounded_Priority_Queues
  child of Ada.Containers A.18.31(2/5)
Bounded_Synchronized_Queues
  child of Ada.Containers A.18.29(2/5)
Bounded_Vectors
  child of Ada.Containers A.18.19(1/3)
C
  child of Interfaces B.3(4/5)
Calendar
  child of Ada 9.6(10/5)
Characters
  child of Ada A.3.1(2/5)
COBOL
  child of Interfaces B.4(7/5)
Command_Line
  child of Ada A.15(3/5)
Complex_Arrays
  child of Ada.Numerics G.3.2(53/2)
Complex_Elementary_Functions
  child of Ada.Numerics G.1.2(9/1)
Complex_Text_IO
  child of Ada G.1.3(9.1/2)
Complex_Types
  child of Ada.Numerics G.1.1(25/1)
Complex_IO
  child of Ada.Text_IO G.1.3(3/5)
  child of Ada.Wide_Text_IO G.1.4(1)
  child of Ada.Wide_Wide_Text_IO G.1.5(1/2)
Constants
  child of Ada.Strings.Maps A.4.6(3/5)
Containers
  child of Ada A.18.1(3/5)
Conversions
  child of Ada.Characters A.3.4(2/5)
  child of Ada.Strings.UTF_Encoding A.4.11(15/5)
Decimal
  child of Ada F.2(2/5)
Decimal_Conversions
  in Interfaces.COBOL B.4(31)
Decimal_IO
  in Ada.Text_IO A.10.1(73)
Decimal_Output
  in Ada.Text_IO.Editing F.3.3(11)
Direct_IO
  child of Ada A.8.4(2/5)
Directories
  child of Ada A.16(3/5)
Discrete_Random
  child of Ada.Numerics A.5.2(17/5)
Dispatching
  child of Ada D.2.1(1.2/5)
Dispatching_Domains
  child of System.Multiprocessors D.16.1(3/5)

Double_Precision_Complex_Types
in Interfaces.Fortran B.5(10.1/5)
Doubly_Linked_Lists
child of Ada.Containers A.18.3(5/5)
Dynamic_Priorities
child of Ada D.5.1(3/5)
EDF
child of Ada.Dispatching D.2.6(9/5)
child of Ada.Synchronous_Task_Control D.10(5.2/5)
Editing
child of Ada.Text_IO F.3.3(3/5)
child of Ada.Wide_Text_IO F.3.4(1)
child of Ada.Wide_Wide_Text_IO F.3.5(1/2)
Elementary_Functions
child of Ada.Numerics A.5.1(9/1)
Enumeration_IO
in Ada.Text_IO A.10.1(79)
Environment_Variables
child of Ada A.17(3/5)
Exceptions
child of Ada 11.4.1(2/5)
Exchange
child of System.Atomic_Operations C.6.2(5/5)
Execution_Time
child of Ada D.14(3/5)
Finalization
child of Ada 7.6(4/5)
Fixed
child of Ada.Strings A.4.3(5/5)
Fixed_Conversions
in Ada.Numerics.Big_Numbers.Big_Reals A.5.7(14/5)
Fixed_IO
in Ada.Text_IO A.10.1(68)
Float_Random
child of Ada.Numerics A.5.2(5/5)
Float_Text_IO
child of Ada A.10.9(33)
Float_Wide_Text_IO
child of Ada A.11(2/2)
Float_Wide_Wide_Text_IO
child of Ada A.11(3/2)
Float_Conversions
in Ada.Numerics.Big_Numbers.Big_Reals A.5.7(13/5)
Float_IO
in Ada.Text_IO A.10.1(63)
Formatting
child of Ada.Calendar 9.6.1(15/5)
Fortran
child of Interfaces B.5(4/5)
Generic_Complex_Arrays
child of Ada.Numerics G.3.2(2/5)
Generic_Complex_Elementary_Functions
child of Ada.Numerics G.1.2(2/5)
Generic_Complex_Types
child of Ada.Numerics G.1.1(2/5)
Generic_Dispatching_Constructor
child of Ada.Tags 3.9(18.2/5)
Generic_Elementary_Functions
child of Ada.Numerics A.5.1(3/5)
Generic_Bounded_Length
in Ada.Strings.Bounded A.4.4(4)
Generic_Keys
in Ada.Containers.Hashed_Sets A.18.8(50/5)
in Ada.Containers.Ordered_Sets A.18.9(62/5)
Generic_Real_Arrays
child of Ada.Numerics G.3.1(2/5)
Generic_Sorting
in Ada.Containers.Doubly_Linked_Lists A.18.3(47/5)
in Ada.Containers.Vectors A.18.2(75/5)
Group_Budgets
child of Ada.Execution_Time D.14.2(3/5)
Handling
child of Ada.Characters A.3.2(2/5)
child of Ada.Wide_Characters A.3.5(3/5)
child of Ada.Wide_Wide_Characters A.3.6(1/3)
Hashed_Maps
child of Ada.Containers A.18.5(2/5)
Hashed_Sets
child of Ada.Containers A.18.8(2/5)
Hierarchical_File_Names
child of Ada.Directories A.16.1(3/5)
child of Ada.Wide_Directories A.16.2(2/5)
child of Ada.Wide_Wide_Directories A.16.2(3/5)
Indefinite_Doubly_Linked_Lists
child of Ada.Containers A.18.12(2/3)
Indefinite_Hashed_Maps
child of Ada.Containers A.18.13(2/3)
Indefinite_Hashed_Sets
child of Ada.Containers A.18.15(2/3)
Indefinite_Holders
child of Ada.Containers A.18.18(5/5)
Indefinite_Multiway_Trees
child of Ada.Containers A.18.17(2/3)
Indefinite_Ordered_Maps
child of Ada.Containers A.18.14(2/3)
Indefinite_Ordered_Sets
child of Ada.Containers A.18.16(2/3)
Indefinite_Vectors
child of Ada.Containers A.18.11(2/3)
Information
child of Ada.Directories A.16(124/2)
child of Ada.Wide_Directories A.16.2(2/5)
child of Ada.Wide_Wide_Directories A.16.2(3/5)
Integer_Text_IO
child of Ada A.10.8(21)
Integer_Wide_Text_IO
child of Ada A.11(2/2)
Integer_Wide_Wide_Text_IO
child of Ada A.11(3/2)
Integer_Arithmetic
child of System.Atomic_Operations C.6.4(3/5)
Integer_IO
in Ada.Text_IO A.10.1(52)
Interfaces B.2(3/5)
Interrupts
child of Ada C.3.2(2/5)
child of Ada.Execution_Time D.14.3(3/5)
IO_Exceptions
child of Ada A.13(3/5)
Iterator_Interfaces
child of Ada 5.5.1(2/5)
Latin_1
child of Ada.Characters A.3.3(3/5)
List_Iterator_Interfaces
in Ada.Containers.Doubly_Linked_Lists A.18.3(9.3/3), A.18.3(51.7/5)
Locales
child of Ada A.19(3/5)

Machine_Code
child of System 13.8(7)
Map_Iterator_Interfaces
in Ada.Containers.Hashed_Maps A.18.5(6.3/5), A.18.5(37.8/5)
in Ada.Containers.Ordered_Maps A.18.6(7.3/5), A.18.6(51.9/5)
Maps
child of Ada.Strings A.4.2(3/5)
Modular_Arithmetic
child of System.Atomic_Operations C.6.5(3/5)
Modular_IO
in Ada.Text_IO A.10.1(57)
Multiprocessors
child of System D.16(3/5)
Multiway_Trees
child of Ada.Containers A.18.10(7/5)
Names
child of Ada.Interrupts C.3.2(12/5)
Non_Preemptive
child of Ada.Dispatching D.2.4(2.2/5)
Numerics
child of Ada A.5(3/5)
Ordered_Maps
child of Ada.Containers A.18.6(2/5)
Ordered_Sets
child of Ada.Containers A.18.9(2/5)
Pointers
child of Interfaces.C B.3.2(4/5)
Real_Arrays
child of Ada.Numerics G.3.1(31/2)
Real_Time
child of Ada D.8(3/5)
Round_Robin
child of Ada.Dispatching D.2.5(4/5)
RPC
child of System E.5(3/5)
Sequential_IO
child of Ada A.8.1(2/5)
Set_Iterator_Interfaces
in Ada.Containers.Hashed_Sets A.18.8(6.3/3), A.18.8(59.7/5)
in Ada.Containers.Ordered_Sets A.18.9(7.3/3), A.18.9(74.7/5)
Signed_Conversions
in Ada.Numerics.Big_Numbers.Big_Integers A.5.6(12/5)
Single_Precision_Complex_Types
in Interfaces.Fortran B.5(8)
Stable
in Ada.Containers.Doubly_Linked_Lists A.18.3(51.1/5)
in Ada.Containers.Hashed_Maps A.18.5(37.2/5)
in Ada.Containers.Hashed_Sets A.18.8(59.1/5)
in Ada.Containers.Multiway_Trees A.18.10(70.1/5)
in Ada.Containers.Ordered_Maps A.18.6(51.3/5)
in Ada.Containers.Ordered_Sets A.18.9(74.1/5)
in Ada.Containers.Vectors A.18.2(79.1/5)
Standard A.1(4/5)
Storage
child of Ada.Streams 13.13.1(12/5)
Storage_Elements
child of System 13.7.1(2/5)
Storage_IO
child of Ada A.9(3/5)
Storage_Pools
child of System 13.11(5/5)
Stream_IO
child of Ada.Streams A.12.1(3/5)
Streams
child of Ada 13.13.1(2/5)
Strings
child of Ada A.4.1(3/5)
child of Ada.Strings.UTF_Encoding A.4.11(22/5)
child of Interfaces.C B.3.1(3/5)
Subpools
child of System.Storage_Pools 13.11.4(3/5)
Synchronized_Queue_Interfaces
child of Ada.Containers A.18.27(3/5)
Synchronous_Barriers
child of Ada D.10.1(3/5)
Synchronous_Task_Control
child of Ada D.10(3/5)
System 13.7(3/5)
Tags
child of Ada 3.9(6/5)
Task_Attributes
child of Ada C.7.2(2/5)
Task_Identification
child of Ada C.7.1(2/5)
Task_Termination
child of Ada C.7.3(2/5)
Test_And_Set
child of System.Atomic_Operations C.6.3(3/5)
Text_Streams
child of Ada.Text_IO A.12.2(3/5)
child of Ada.Wide_Text_IO A.12.3(3/5)
child of Ada.Wide_Wide_Text_IO A.12.4(3/5)
Text_Buffers
child of Ada.Strings A.4.12(3/5)
Text_IO
child of Ada A.10.1(2/5)
Time_Zones
child of Ada.Calendar 9.6.1(2/5)
Timers
child of Ada.Execution_Time D.14.1(3/5)
Timing_Events
child of Ada.Real_Time D.15(3/5)
Tree_Iterator_Interfaces
in Ada.Containers.Multiway_Trees A.18.10(13/3), A.18.10(70.7/5)
Unbounded
child of Ada.Streams.Storage 13.13.1(18/5)
child of Ada.Strings A.4.5(3/5)
child of Ada.Strings.Text_Buffers A.4.12(18/5)
Unbounded_IO
child of Ada.Text_IO A.10.12(3/5)
Unbounded_Priority_Queues
child of Ada.Containers A.18.30(2/5)
Unbounded_Synchronized_Queues
child of Ada.Containers A.18.28(2/5)
Unsigned_Conversions
in Ada.Numerics.Big_Numbers.Big_Integers A.5.6(13/5)
UTF_Encoding
child of Ada.Strings A.4.11(3/5)
Vector_Iterator_Interfaces
in Ada.Containers.Vectors A.18.2(11.3/5), A.18.2(79.7/5)
Vectors
child of Ada.Containers A.18.2(6/5)

Wide_Bounded
  child of Ada.Strings A.4.7(1/3)
Wide_Constants
  child of Ada.Strings.Wide_Maps A.4.7(1/3), A.4.8(28/2)
Wide_Equal_Case_Insensitive
  child of Ada.Strings A.4.7(1/3)
Wide_Fixed
  child of Ada.Strings A.4.7(1/3)
Wide_Hash
  child of Ada.Strings A.4.7(1/3)
Wide_Hash_Case_Insensitive
  child of Ada.Strings A.4.7(1/3)
Wide_Maps
  child of Ada.Strings A.4.7(3/5)
Wide_Text_IO
  child of Ada A.11(2/2)
Wide_Unbounded
  child of Ada.Strings A.4.7(1/3)
Wide_Bounded_IO
  child of Ada.Wide_Text_IO A.11(4/3)
Wide_Characters
  child of Ada A.3.1(4/5)
Wide_Command_Line
  child of Ada A.15.1(2/5)
Wide_Directories
  child of Ada A.16.2(2/5)
Wide_Environment_Variables
  child of Ada A.17.1(2/5)
Wide_File_Names
  in Ada.Direct_IO A.8.4(18.1/5)
  in Ada.Sequential_IO A.8.1(15.1/5)
  in Ada.Streams.Stream_IO A.12.1(26.1/5)
  in Ada.Text_IO A.10.1(85.1/5)
Wide_Strings
  child of Ada.Strings.UTF_Encoding A.4.11(30/5)
Wide_Unbounded_IO
  child of Ada.Wide_Text_IO A.11(5/3)
Wide_Wide_Constants
  child of Ada.Strings.Wide_Wide_Maps A.4.8(1/3)
Wide_Wide_Equal_Case_Insensitive
  child of Ada.Strings A.4.8(1/3)
Wide_Wide_Hash
  child of Ada.Strings A.4.8(1/3)
Wide_Wide_Hash_Case_Insensitive
  child of Ada.Strings A.4.8(1/3)
Wide_Wide_Text_IO
  child of Ada A.11(3/2)
Wide_Wide_Bounded
  child of Ada.Strings A.4.8(1/3)
Wide_Wide_Bounded_IO
  child of Ada.Wide_Wide_Text_IO A.11(4/3)
Wide_Wide_Characters
  child of Ada A.3.1(6/5)
Wide_Wide_Command_Line
  child of Ada A.15.1(3/5)
Wide_Wide_Directories
  child of Ada A.16.2(3/5)
Wide_Wide_Environment_Variables
  child of Ada A.17.1(3/5)
Wide_Wide_File_Names
  in Ada.Direct_IO A.8.4(18.8/5)
  in Ada.Sequential_IO A.8.1(15.8/5)
  in Ada.Streams.Stream_IO A.12.1(26.8/5)
  in Ada.Text_IO A.10.1(85.8/5)
Wide_Wide_Fixed
  child of Ada.Strings A.4.8(1/3)
Wide_Wide_Maps
  child of Ada.Strings A.4.8(3/5)
Wide_Wide_Strings
  child of Ada.Strings.UTF_Encoding A.4.11(38/5)
Wide_Wide_Unbounded
  child of Ada.Strings A.4.8(1/3)
Wide_Wide_Unbounded_IO
  child of Ada.Wide_Wide_Text_IO A.11(5/3)

Q.2 Language-Defined Types and Subtypes

1/3 {*AI95-00440-01*} {*AI05-0299-1*} This subclause lists all language-defined types and subtypes.

Address
  in System 13.7(12)
Alignment
  in Ada.Strings A.4.1(6)
Alphanumeric
  in Interfaces.COBOL B.4(16/3)
Any_Priority *subtype of* Integer
  in System 13.7(16)
Attribute_Handle
  in Ada.Task_Attributes C.7.2(3)
Barrier_Limit *subtype of* Positive
  in Ada.Synchronous_Barriers D.10.1(4/3)
Big_Integer
  in Ada.Numerics.Big_Numbers.Big_Integers A.5.6(3/5)
Big_Natural *subtype of* Big_Integer
  in Ada.Numerics.Big_Numbers.Big_Integers A.5.6(9/5)
Big_Positive *subtype of* Big_Integer
  in Ada.Numerics.Big_Numbers.Big_Integers A.5.6(8/5)
Big_Real
  in Ada.Numerics.Big_Numbers.Big_Reals A.5.7(3/5)
Binary
  in Interfaces.COBOL B.4(10)
Binary_Format
  in Interfaces.COBOL B.4(24)
Bit_Order
  in System 13.7(15/2)
Boolean
  in Standard A.1(5)
Bounded_String
  in Ada.Strings.Bounded A.4.4(6)
Buffer_Type
  in Ada.Strings.Text_Buffers.Bounded A.4.12(27/5)
  in Ada.Strings.Text_Buffers.Unbounded A.4.12(19/5)
Buffer_Type *subtype of* Storage_Array
  in Ada.Storage_IO A.9(4)
Byte
  in Interfaces.COBOL B.4(29/3)

Byte_Array
in Interfaces.COBOL B.4(29/3)
C_bool
in Interfaces.C B.3(13.1/5)
C_float
in Interfaces.C B.3(15)
Cause_Of_Termination
in Ada.Task_Termination C.7.3(3/2)
char
in Interfaces.C B.3(19)
char16_array
in Interfaces.C B.3(39.5/3)
char16_t
in Interfaces.C B.3(39.2/2)
char32_array
in Interfaces.C B.3(39.14/3)
char32_t
in Interfaces.C B.3(39.11/2)
char_array
in Interfaces.C B.3(23/3)
char_array_access
in Interfaces.C.Strings B.3.1(4)
Character
in Standard A.1(35/3)
Character_Mapping
in Ada.Strings.Maps A.4.2(20/5)
Character_Mapping_Function
in Ada.Strings.Maps A.4.2(25)
Character_Range
in Ada.Strings.Maps A.4.2(6)
Character_Ranges
in Ada.Strings.Maps A.4.2(7)
Character_Sequence *subtype of* String
in Ada.Strings.Maps A.4.2(16)
Character_Set
in Ada.Strings.Maps A.4.2(4/5)
in Interfaces.Fortran B.5(11)
chars_ptr
in Interfaces.C.Strings B.3.1(5/5)
chars_ptr_array
in Interfaces.C.Strings B.3.1(6/2)
Chunk_Index *subtype of* Positive
in Ada.Iterator_Interfaces 5.5.1(4.2/5)
COBOL_Character
in Interfaces.COBOL B.4(13)
Complex
in Ada.Numerics.Generic_Complex_Types G.1.1(3)
in Interfaces.Fortran B.5(9)
Complex_Matrix
in Ada.Numerics.Generic_Complex_Arrays G.3.2(4/2)
Complex_Vector
in Ada.Numerics.Generic_Complex_Arrays G.3.2(4/2)
Constant_Reference_Type
in Ada.Containers.Doubly_Linked_Lists A.18.3(51.10/5)
in Ada.Containers.Hashed_Maps A.18.5(37.11/5)
in Ada.Containers.Hashed_Sets A.18.8(59.10/5)
in Ada.Containers.Indefinite_Holders A.18.18(16/5)
in Ada.Containers.Multiway_Trees A.18.10(70.10/5)
in Ada.Containers.Ordered_Maps A.18.6(51.12/5)
in Ada.Containers.Ordered_Sets A.18.9(74.10/5)
in Ada.Containers.Vectors A.18.2(79.10/5)
Controlled
in Ada.Finalization 7.6(5/5)
Count
in Ada.Direct_IO A.8.4(4)
in Ada.Streams.Stream_IO A.12.1(7)
in Ada.Text_IO A.10.1(5)
Count_Type
in Ada.Containers A.18.1(5/2)
Country_Code
in Ada.Locales A.19(4/4)
CPU *subtype of* CPU_Range
in System.Multiprocessors D.16(4/3)
CPU_Range
in System.Multiprocessors D.16(4/3)
CPU_Set
in System.Multiprocessors.Dispatching_Domains D.16.1(9.1/4)
CPU_Time
in Ada.Execution_Time D.14(4/2)
Cursor
in Ada.Containers.Doubly_Linked_Lists A.18.3(7/5), A.18.3(51.3/5)
in Ada.Containers.Hashed_Maps A.18.5(4/5), A.18.5(37.4/5)
in Ada.Containers.Hashed_Sets A.18.8(4/5), A.18.8(59.3/5)
in Ada.Containers.Multiway_Trees A.18.10(9/5), A.18.10(70.3/5)
in Ada.Containers.Ordered_Maps A.18.6(5/5), A.18.6(51.5/5)
in Ada.Containers.Ordered_Sets A.18.9(5/5), A.18.9(74.3/5)
in Ada.Containers.Vectors A.18.2(9/5), A.18.2(79.3/5)
Day_Count
in Ada.Calendar.Arithmetic 9.6.1(10/2)
Day_Duration *subtype of* Duration
in Ada.Calendar 9.6(11/2)
Day_Name
in Ada.Calendar.Formatting 9.6.1(17/2)
Day_Number *subtype of* Integer
in Ada.Calendar 9.6(11/2)
Deadline *subtype of* Time
in Ada.Dispatching.EDF D.2.6(9/5)
Decimal_Element
in Interfaces.COBOL B.4(12/3)
Direction
in Ada.Strings A.4.1(6)
Directory_Entry_Type
in Ada.Directories A.16(29/2)
Dispatching_Domain
in System.Multiprocessors.Dispatching_Domains D.16.1(5/3)
Display_Format
in Interfaces.COBOL B.4(22)
double
in Interfaces.C B.3(16)
Double_Complex
in Interfaces.Fortran B.5(10.2/5)
Double_Imaginary *subtype of* Imaginary
in Interfaces.Fortran B.5(10.3/5)
Double_Precision
in Interfaces.Fortran B.5(6)
Duration
in Standard A.1(43)
Encoding_Scheme
in Ada.Strings.UTF_Encoding A.4.11(4/3)
Exception_Id
in Ada.Exceptions 11.4.1(2/5)

Exception_Occurrence
  in Ada.Exceptions 11.4.1(3/5)
Exception_Occurrence_Access
  in Ada.Exceptions 11.4.1(3/5)
Exit_Status
  in Ada.Command_Line A.15(7)
Extended_Index *subtype of* Index_Type'Base
  in Ada.Containers.Vectors A.18.2(7/2)
Field *subtype of* Integer
  in Ada.Numerics.Big_Numbers.Big_Integers A.5.5(3/5)
  in Ada.Text_IO A.10.1(6)
File_Access
  in Ada.Text_IO A.10.1(18)
File_Kind
  in Ada.Directories A.16(22/2)
File_Mode
  in Ada.Direct_IO A.8.4(4)
  in Ada.Sequential_IO A.8.1(4)
  in Ada.Streams.Stream_IO A.12.1(6)
  in Ada.Text_IO A.10.1(4)
File_Size
  in Ada.Directories A.16(23/2)
File_Type
  in Ada.Direct_IO A.8.4(3)
  in Ada.Sequential_IO A.8.1(3)
  in Ada.Streams.Stream_IO A.12.1(5/5)
  in Ada.Text_IO A.10.1(3)
Filter_Type
  in Ada.Directories A.16(30/2)
Float
  in Standard A.1(21)
Floating
  in Interfaces.COBOL B.4(9)
Fortran_Character
  in Interfaces.Fortran B.5(12/3)
Fortran_Integer
  in Interfaces.Fortran B.5(5)
Forward_Iterator
  in Ada.Iterator_Interfaces 5.5.1(3/3)
Generator
  in Ada.Numerics.Discrete_Random A.5.2(19)
  in Ada.Numerics.Float_Random A.5.2(7)
Group_Budget
  in Ada.Execution_Time.Group_Budgets D.14.2(4/3)
Group_Budget_Handler
  in Ada.Execution_Time.Group_Budgets D.14.2(5/5)
Hash_Type
  in Ada.Containers A.18.1(4/2)
Holder
  in Ada.Containers.Indefinite_Holders A.18.18(6/5)
Hour_Number *subtype of* Natural
  in Ada.Calendar.Formatting 9.6.1(20/2)
Imaginary
  in Ada.Numerics.Generic_Complex_Types G.1.1(4/5)
Imaginary *subtype of* Imaginary
  in Interfaces.Fortran B.5(10)
int
  in Interfaces.C B.3(7)
Integer
  in Standard A.1(12)
Integer_Address
  in System.Storage_Elements 13.7.1(10/3)
Interrupt_Id
  in Ada.Interrupts C.3.2(2/5)
Interrupt_Priority *subtype of* Any_Priority
  in System 13.7(16)
ISO_646 *subtype of* Character
  in Ada.Characters.Handling A.3.2(9)
Language_Code
  in Ada.Locales A.19(4/4)
Leap_Seconds_Count *subtype of* Integer
  in Ada.Calendar.Arithmetic 9.6.1(11/2)
Length_Range *subtype of* Natural
  in Ada.Strings.Bounded A.4.4(8)
Limited_Controlled
  in Ada.Finalization 7.6(7/5)
List
  in Ada.Containers.Doubly_Linked_Lists A.18.3(6/5),
    A.18.3(51.2/5)
Logical
  in Interfaces.Fortran B.5(7)
long
  in Interfaces.C B.3(7)
Long_Binary
  in Interfaces.COBOL B.4(10)
long_double
  in Interfaces.C B.3(17)
Long_Floating
  in Interfaces.COBOL B.4(9)
Map
  in Ada.Containers.Hashed_Maps A.18.5(3/5), A.18.5(37.3/5)
  in Ada.Containers.Ordered_Maps A.18.6(4/5),
    A.18.6(51.4/5)
Membership
  in Ada.Strings A.4.1(6)
Minute_Number *subtype of* Natural
  in Ada.Calendar.Formatting 9.6.1(20/2)
Month_Number *subtype of* Integer
  in Ada.Calendar 9.6(11/2)
Name
  in System 13.7(4)
Name_Case_Kind
  in Ada.Directories A.16(20.1/3)
Natural *subtype of* Integer
  in Standard A.1(13)
Number_Base *subtype of* Integer
  in Ada.Numerics.Big_Numbers.Big_Integers A.5.5(3/5)
  in Ada.Text_IO A.10.1(6)
Numeric
  in Interfaces.COBOL B.4(20/3)
Packed_Decimal
  in Interfaces.COBOL B.4(12/3)
Packed_Format
  in Interfaces.COBOL B.4(26)
Parallel_Iterator
  in Ada.Iterator_Interfaces 5.5.1(4.1/5)
Parallel_Reversible_Iterator
  in Ada.Iterator_Interfaces 5.5.1(4.8/5)
Parameterless_Handler
  in Ada.Interrupts C.3.2(2/5)
Params_Stream_Type
  in System.RPC E.5(6)
Partition_Id
  in System.RPC E.5(4)
Picture
  in Ada.Text_IO.Editing F.3.3(4)
plain_char
  in Interfaces.C B.3(11)

Pointer
in Interfaces.C.Pointers B.3.2(5)
Positive *subtype of* Integer
in Standard A.1(13)
Positive_Count *subtype of* Count
in Ada.Direct_IO A.8.4(4)
in Ada.Streams.Stream_IO A.12.1(7)
in Ada.Text_IO A.10.1(5)
Priority *subtype of* Any_Priority
in System 13.7(16)
ptrdiff_t
in Interfaces.C B.3(12)
Queue
in Ada.Containers.Bounded_Priority_Queues A.18.31(4/3)
in Ada.Containers.Bounded_Synchronized_Queues A.18.29(4/3)
in Ada.Containers.Synchronized_Queue_Interfaces A.18.27(4/3)
in Ada.Containers.Unbounded_Priority_Queues A.18.30(4/3)
in Ada.Containers.Unbounded_Synchronized_Queues A.18.28(4/3)
Real
in Interfaces.Fortran B.5(6)
Real_Matrix
in Ada.Numerics.Generic_Real_Arrays G.3.1(4/2)
Real_Vector
in Ada.Numerics.Generic_Real_Arrays G.3.1(4/2)
Reference_Type
in Ada.Containers.Doubly_Linked_Lists A.18.3(17.2/5), A.18.3(51.11/5)
in Ada.Containers.Hashed_Maps A.18.5(17.2/5), A.18.5(37.12/5)
in Ada.Containers.Hashed_Sets A.18.8(58.1/5)
in Ada.Containers.Indefinite_Holders A.18.18(17/5)
in Ada.Containers.Multiway_Trees A.18.10(29/5), A.18.10(70.11/5)
in Ada.Containers.Ordered_Maps A.18.6(16.2/5), A.18.6(51.13/5)
in Ada.Containers.Ordered_Sets A.18.9(73.1/5)
in Ada.Containers.Vectors A.18.2(34.2/5), A.18.2(79.11/5)
Relative_Deadline *subtype of* Time_Span
in Ada.Dispatching.EDF D.2.6(9/5)
Reversible_Iterator
in Ada.Iterator_Interfaces 5.5.1(4/3)
Root_Buffer_Type
in Ada.Strings.Text_Buffers A.4.12(6/5)
Root_Storage_Pool
in System.Storage_Pools 13.11(6/5)
Root_Storage_Pool_With_Subpools
in System.Storage_Pools.Subpools 13.11.4(4/5)
Root_Stream_Type
in Ada.Streams 13.13.1(3/2)
Root_Subpool
in System.Storage_Pools.Subpools 13.11.4(5/5)
RPC_Receiver
in System.RPC E.5(11)
Search_Type
in Ada.Directories A.16(31/2)
Second_Duration *subtype of* Day_Duration
in Ada.Calendar.Formatting 9.6.1(20/2)
Second_Number *subtype of* Natural
in Ada.Calendar.Formatting 9.6.1(20/2)
Seconds_Count
in Ada.Real_Time D.8(15)
Set
in Ada.Containers.Hashed_Sets A.18.8(3/5), A.18.8(59.2/5)
in Ada.Containers.Ordered_Sets A.18.9(4/5), A.18.9(74.2/5)
short
in Interfaces.C B.3(7)
signed_char
in Interfaces.C B.3(8)
size_t
in Interfaces.C B.3(13)
State
in Ada.Numerics.Discrete_Random A.5.2(23)
in Ada.Numerics.Float_Random A.5.2(11)
Storage_Array
in System.Storage_Elements 13.7.1(5)
Storage_Count *subtype of* Storage_Offset
in System.Storage_Elements 13.7.1(4)
Storage_Element
in System.Storage_Elements 13.7.1(5)
Storage_Offset
in System.Storage_Elements 13.7.1(3)
Storage_Stream_Type
in Ada.Streams.Storage 13.13.1(13/5)
Stream_Access
in Ada.Streams.Stream_IO A.12.1(4)
in Ada.Text_IO.Text_Streams A.12.2(3/5)
in Ada.Wide_Text_IO.Text_Streams A.12.3(3/5)
in Ada.Wide_Wide_Text_IO.Text_Streams A.12.4(3/5)
Stream_Element
in Ada.Streams 13.13.1(4/1)
Stream_Element_Array
in Ada.Streams 13.13.1(4/1)
Stream_Element_Count *subtype of* Stream_Element_Offset
in Ada.Streams 13.13.1(4/1)
Stream_Element_Offset
in Ada.Streams 13.13.1(4/1)
Stream_Type
in Ada.Streams.Storage.Bounded 13.13.1(27/5)
in Ada.Streams.Storage.Unbounded 13.13.1(19/5)
String
in Standard A.1(37/3)
String_Access
in Ada.Strings.Unbounded A.4.5(7)
Subpool_Handle
in System.Storage_Pools.Subpools 13.11.4(6/3)
Suspension_Object
in Ada.Synchronous_Task_Control D.10(4/5)
Synchronous_Barrier
in Ada.Synchronous_Barriers D.10.1(5/3)
Tag
in Ada.Tags 3.9(6/5)
Tag_Array
in Ada.Tags 3.9(7.3/2)
Task_Array
in Ada.Execution_Time.Group_Budgets D.14.2(6/2)
Task_Id
in Ada.Task_Identification C.7.1(2/5)
Termination_Handler
in Ada.Task_Termination C.7.3(4/2)
Test_And_Set_Flag
in System.Atomic_Operations.Test_And_Set C.6.3(4/5)
Text_Buffer_Count
in Ada.Strings.Text_Buffers A.4.12(4/5)

Time
  in Ada.Calendar 9.6(10/5)
  in Ada.Real_Time D.8(4)
Time_Offset
  in Ada.Calendar.Time_Zones 9.6.1(4/2)
Time_Span
  in Ada.Real_Time D.8(5)
Timer
  in Ada.Execution_Time.Timers D.14.1(4/2)
Timer_Handler
  in Ada.Execution_Time.Timers D.14.1(5/5)
Timing_Event
  in Ada.Real_Time.Timing_Events D.15(4/5)
Timing_Event_Handler
  in Ada.Real_Time.Timing_Events D.15(4/5)
Tree
  in Ada.Containers.Multiway_Trees A.18.10(8/5), A.18.10(70.2/5)
Trim_End
  in Ada.Strings A.4.1(6)
Truncation
  in Ada.Strings A.4.1(6)
Type_Set
  in Ada.Text_IO A.10.1(7)
Unbounded_String
  in Ada.Strings.Unbounded A.4.5(4/5)
Uniformly_Distributed *subtype of* Float
  in Ada.Numerics.Float_Random A.5.2(8/5)
unsigned
  in Interfaces.C B.3(9)
unsigned_char
  in Interfaces.C B.3(10)
unsigned_long
  in Interfaces.C B.3(9)
unsigned_short
  in Interfaces.C B.3(9)
UTF_16_Wide_String *subtype of* Wide_String
  in Ada.Strings.UTF_Encoding A.4.11(7/3)
UTF_8_String *subtype of* String
  in Ada.Strings.UTF_Encoding A.4.11(6/3)
UTF_String *subtype of* String
  in Ada.Strings.UTF_Encoding A.4.11(5/3)
Valid_Big_Integer *subtype of* Big_Integer
  in Ada.Numerics.Big_Numbers.Big_Integers A.5.6(5/5)
Valid_Big_Real *subtype of* Big_Real
  in Ada.Numerics.Big_Numbers.Big_Reals A.5.7(5/5)
Vector
  in Ada.Containers.Vectors A.18.2(8/5), A.18.2(79.2/5)
wchar_array
  in Interfaces.C B.3(33/3)
wchar_t
  in Interfaces.C B.3(30/1)
Wide_Character
  in Standard A.1(36.1/5)
Wide_Character_Mapping
  in Ada.Strings.Wide_Maps A.4.7(20/5)
Wide_Character_Mapping_Function
  in Ada.Strings.Wide_Maps A.4.7(26)
Wide_Character_Range
  in Ada.Strings.Wide_Maps A.4.7(6)
Wide_Character_Ranges
  in Ada.Strings.Wide_Maps A.4.7(7)
Wide_Character_Sequence *subtype of* Wide_String
  in Ada.Strings.Wide_Maps A.4.7(16)
Wide_Character_Set
  in Ada.Strings.Wide_Maps A.4.7(4/5)
Wide_String
  in Standard A.1(41/3)
Wide_Wide_Character
  in Standard A.1(36.2/5)
Wide_Wide_Character_Mapping
  in Ada.Strings.Wide_Wide_Maps A.4.8(20/5)
Wide_Wide_Character_Mapping_Function
  in Ada.Strings.Wide_Wide_Maps A.4.8(26/2)
Wide_Wide_Character_Range
  in Ada.Strings.Wide_Wide_Maps A.4.8(6/2)
Wide_Wide_Character_Ranges
  in Ada.Strings.Wide_Wide_Maps A.4.8(7/2)
Wide_Wide_Character_Sequence *subtype of* Wide_Wide_String
  in Ada.Strings.Wide_Wide_Maps A.4.8(16/2)
Wide_Wide_Character_Set
  in Ada.Strings.Wide_Wide_Maps A.4.8(4/5)
Wide_Wide_String
  in Standard A.1(42.1/3)
Year_Number *subtype of* Integer
  in Ada.Calendar 9.6(11/2)

Q.3 Language-Defined Subprograms

1/3 {*AI95-00440-01*} {*AI05-0299-1*} This subclause lists all language-defined subprograms.

Abort_Task *in* Ada.Task_Identification C.7.1(3/5)
Activation_Is_Complete
  in Ada.Task_Identification C.7.1(4/3)
Actual_Quantum
  in Ada.Dispatching.Round_Robin D.2.5(4/5)
Ada.Unchecked_Deallocate_Subpool
  child of Ada 13.11.5(3/5)
Add
  in Ada.Execution_Time.Group_Budgets D.14.2(9/2)
Add_Task
  in Ada.Execution_Time.Group_Budgets D.14.2(8/2)
Adjust *in* Ada.Finalization 7.6(6/2)
Allocate
  in System.Storage_Pools 13.11(7)
  in System.Storage_Pools.Subpools 13.11.4(14/3)
Allocate_From_Subpool
  in System.Storage_Pools.Subpools 13.11.4(11/5)
Ancestor_Find
  in Ada.Containers.Multiway_Trees A.18.10(40.1/5), A.18.10(40/5)
Append
  in Ada.Containers.Doubly_Linked_Lists A.18.3(23.1/5), A.18.3(23/5)
  in Ada.Containers.Vectors A.18.2(47.1/5), A.18.2(47/5)

in Ada.Strings.Bounded A.4.4(13), A.4.4(14), A.4.4(15), A.4.4(16), A.4.4(17), A.4.4(18), A.4.4(19), A.4.4(20)
in Ada.Strings.Unbounded A.4.5(12), A.4.5(13), A.4.5(14)
Append_Child
in Ada.Containers.Multiway_Trees A.18.10(52/5)
Append_Vector
in Ada.Containers.Vectors A.18.2(46/5)
Arccos
in Ada.Numerics.Generic_Complex_Elementary_Functions G.1.2(5)
in Ada.Numerics.Generic_Elementary_Functions A.5.1(6)
Arccosh
in Ada.Numerics.Generic_Complex_Elementary_Functions G.1.2(7)
in Ada.Numerics.Generic_Elementary_Functions A.5.1(7)
Arccot
in Ada.Numerics.Generic_Complex_Elementary_Functions G.1.2(5)
in Ada.Numerics.Generic_Elementary_Functions A.5.1(6)
Arccoth
in Ada.Numerics.Generic_Complex_Elementary_Functions G.1.2(7)
in Ada.Numerics.Generic_Elementary_Functions A.5.1(7)
Arcsin
in Ada.Numerics.Generic_Complex_Elementary_Functions G.1.2(5)
in Ada.Numerics.Generic_Elementary_Functions A.5.1(6)
Arcsinh
in Ada.Numerics.Generic_Complex_Elementary_Functions G.1.2(7)
in Ada.Numerics.Generic_Elementary_Functions A.5.1(7)
Arctan
in Ada.Numerics.Generic_Complex_Elementary_Functions G.1.2(5)
in Ada.Numerics.Generic_Elementary_Functions A.5.1(6)
Arctanh
in Ada.Numerics.Generic_Complex_Elementary_Functions G.1.2(7)
in Ada.Numerics.Generic_Elementary_Functions A.5.1(7)
Argument
in Ada.Command_Line A.15(5)
in Ada.Numerics.Generic_Complex_Arrays G.3.2(10/2), G.3.2(31/2)
in Ada.Numerics.Generic_Complex_Types G.1.1(10)
Argument_Count *in* Ada.Command_Line A.15(4)
Assert *in* Ada.Assertions 11.4.2(14/2)
Assign
in Ada.Containers.Doubly_Linked_Lists A.18.3(17.5/5), A.18.3(51.8/5)
in Ada.Containers.Hashed_Maps A.18.5(17.7/5), A.18.5(37.9/5)
in Ada.Containers.Hashed_Sets A.18.8(17.4/5), A.18.8(59.8/5)
in Ada.Containers.Indefinite_Holders A.18.18(20/5)
in Ada.Containers.Multiway_Trees A.18.10(32/5), A.18.10(70.8/5)
in Ada.Containers.Ordered_Maps A.18.6(16.7/5), A.18.6(51.10/5)
in Ada.Containers.Ordered_Sets A.18.9(16.4/5), A.18.9(74.8/5)
in Ada.Containers.Vectors A.18.2(34.7/5), A.18.2(79.8/5)
Assign_Task
in System.Multiprocessors.Dispatching_Domains D.16.1(11/3)
Atomic_Add
in System.Atomic_Operations.Integer_Arithmetic C.6.4(4/5)
in System.Atomic_Operations.Modular_Arithmetic C.6.5(4/5)
Atomic_Clear
in System.Atomic_Operations.Test_And_Set C.6.3(6/5)
Atomic_Compare_And_Exchange
in System.Atomic_Operations.Exchange C.6.2(7/5)
Atomic_Exchange
in System.Atomic_Operations.Exchange C.6.2(6/5)
Atomic_Fetch_And_Add
in System.Atomic_Operations.Integer_Arithmetic C.6.4(6/5)
in System.Atomic_Operations.Modular_Arithmetic C.6.5(6/5)
Atomic_Fetch_And_Subtract
in System.Atomic_Operations.Integer_Arithmetic C.6.4(7/5)
in System.Atomic_Operations.Modular_Arithmetic C.6.5(7/5)
Atomic_Subtract
in System.Atomic_Operations.Integer_Arithmetic C.6.4(5/5)
in System.Atomic_Operations.Modular_Arithmetic C.6.5(5/5)
Atomic_Test_And_Set
in System.Atomic_Operations.Test_And_Set C.6.3(5/5)
Attach_Handler *in* Ada.Interrupts C.3.2(7)
Base_Name *in* Ada.Directories A.16(19/5)
Blank_When_Zero
in Ada.Text_IO.Editing F.3.3(7)
Bounded_Slice *in* Ada.Strings.Bounded A.4.4(28.1/2), A.4.4(28.2/2)
Budget_Has_Expired
in Ada.Execution_Time.Group_Budgets D.14.2(9/2)
Budget_Remaining
in Ada.Execution_Time.Group_Budgets D.14.2(9/2)
Cancel_Handler
in Ada.Execution_Time.Group_Budgets D.14.2(10/2)
in Ada.Execution_Time.Timers D.14.1(7/2)
in Ada.Real_Time.Timing_Events D.15(5/2)
Capacity
in Ada.Containers.Hashed_Maps A.18.5(8/5)
in Ada.Containers.Hashed_Sets A.18.8(10/5)
in Ada.Containers.Vectors A.18.2(19/5)
Ceiling
in Ada.Containers.Ordered_Maps A.18.6(41/5)
in Ada.Containers.Ordered_Sets A.18.9(51/2), A.18.9(71/5)
Character_Set_Version
in Ada.Wide_Characters.Handling A.3.5(4/3)
Child_Count
in Ada.Containers.Multiway_Trees A.18.10(46.1/5), A.18.10(46/5)
Child_Depth
in Ada.Containers.Multiway_Trees A.18.10(47.1/5), A.18.10(47/5)
Chunk_Count
in Ada.Iterator_Interfaces 5.5.1(4.5/5)
Clear
in Ada.Containers.Doubly_Linked_Lists A.18.3(13/5)
in Ada.Containers.Hashed_Maps A.18.5(12/5)
in Ada.Containers.Hashed_Sets A.18.8(14/5)
in Ada.Containers.Indefinite_Holders A.18.18(11/5)
in Ada.Containers.Multiway_Trees A.18.10(23/5)
in Ada.Containers.Ordered_Maps A.18.6(11/5)
in Ada.Containers.Ordered_Sets A.18.9(13/5)
in Ada.Containers.Vectors A.18.2(24/5)

in Ada.Environment_Variables A.17(7/2)
in Ada.Streams.Storage 13.13.1(15/5)
in Ada.Streams.Storage.Bounded 13.13.1(31/5)
in Ada.Streams.Storage.Unbounded 13.13.1(23/5)
Clock
in Ada.Calendar 9.6(12)
in Ada.Execution_Time D.14(5/2)
in Ada.Execution_Time.Interrupts D.14.3(3/5)
in Ada.Real_Time D.8(6)
Clock_For_Interrupts
in Ada.Execution_Time D.14(9.3/3)
Close
in Ada.Direct_IO A.8.4(8)
in Ada.Sequential_IO A.8.1(8)
in Ada.Streams.Stream_IO A.12.1(10)
in Ada.Text_IO A.10.1(11)
Col *in* Ada.Text_IO A.10.1(37/5)
Command_Name *in* Ada.Command_Line A.15(6)
Compose
in Ada.Directories A.16(20/5)
in Ada.Directories.Hierarchical_File_Names A.16.1(14/3)
Compose_From_Cartesian
in Ada.Numerics.Generic_Complex_Arrays G.3.2(9/2), G.3.2(29/2)
in Ada.Numerics.Generic_Complex_Types G.1.1(8)
Compose_From_Polar
in Ada.Numerics.Generic_Complex_Arrays G.3.2(11/2), G.3.2(32/2)
in Ada.Numerics.Generic_Complex_Types G.1.1(11)
Conjugate
in Ada.Numerics.Generic_Complex_Arrays G.3.2(13/2), G.3.2(34/2)
in Ada.Numerics.Generic_Complex_Types G.1.1(12), G.1.1(15)
Constant_Reference
in Ada.Containers.Doubly_Linked_Lists A.18.3(17.3/5)
in Ada.Containers.Hashed_Maps A.18.5(17.3/5), A.18.5(17.5/5)
in Ada.Containers.Hashed_Sets A.18.8(17.3/5), A.18.8(58.3/5)
in Ada.Containers.Indefinite_Holders A.18.18(18/5)
in Ada.Containers.Multiway_Trees A.18.10(30/5)
in Ada.Containers.Ordered_Maps A.18.6(16.3/5), A.18.6(16.5/5)
in Ada.Containers.Ordered_Sets A.18.9(16.3/5), A.18.9(73.3/5)
in Ada.Containers.Vectors A.18.2(34.3/5), A.18.2(34.5/5)
Containing_Directory
in Ada.Directories A.16(17/5)
in Ada.Directories.Hierarchical_File_Names A.16.1(11/3)
Contains
in Ada.Containers.Doubly_Linked_Lists A.18.3(43/2)
in Ada.Containers.Hashed_Maps A.18.5(32/2)
in Ada.Containers.Hashed_Sets A.18.8(44/2), A.18.8(57/2)
in Ada.Containers.Multiway_Trees A.18.10(41/3)
in Ada.Containers.Ordered_Maps A.18.6(42/2)
in Ada.Containers.Ordered_Sets A.18.9(52/2), A.18.9(72/2)
in Ada.Containers.Vectors A.18.2(71/2)
Continue
in Ada.Asynchronous_Task_Control D.11(3/5)
Convert
in Ada.Strings.UTF_Encoding.Conversions A.4.11(16/3), A.4.11(17/3), A.4.11(18/3), A.4.11(19/3), A.4.11(20/3)
Copy
in Ada.Containers.Bounded_Doubly_Linked_Lists A.18.20(10/5)
in Ada.Containers.Bounded_Hashed_Maps A.18.21(13/5)
in Ada.Containers.Bounded_Hashed_Sets A.18.23(13/5)
in Ada.Containers.Bounded_Ordered_Maps A.18.22(10/5)
in Ada.Containers.Bounded_Ordered_Sets A.18.24(10/5)
in Ada.Containers.Doubly_Linked_Lists A.18.3(17.6/5), A.18.3(51.9/5)
in Ada.Containers.Hashed_Maps A.18.5(17.8/5), A.18.5(37.10/5)
in Ada.Containers.Hashed_Sets A.18.8(17.5/5), A.18.8(59.9/5)
in Ada.Containers.Indefinite_Holders A.18.18(21/5)
in Ada.Containers.Multiway_Trees A.18.10(33/5), A.18.10(70.9/5)
in Ada.Containers.Ordered_Maps A.18.6(16.8/5), A.18.6(51.11/5)
in Ada.Containers.Ordered_Sets A.18.9(16.5/5), A.18.9(74.9/5)
in Ada.Containers.Vectors A.18.2(34.8/5), A.18.2(79.9/5)
Copy_Array *in* Interfaces.C.Pointers B.3.2(15)
Copy_File *in* Ada.Directories A.16(13/2)
Copy_Local_Subtree
in Ada.Containers.Multiway_Trees A.18.10(54.1/5)
Copy_Subtree
in Ada.Containers.Multiway_Trees A.18.10(54.2/5), A.18.10(54/5)
Copy_Terminated_Array
in Interfaces.C.Pointers B.3.2(14)
Cos
in Ada.Numerics.Generic_Complex_Elementary_Functions G.1.2(4)
in Ada.Numerics.Generic_Elementary_Functions A.5.1(5)
Cosh
in Ada.Numerics.Generic_Complex_Elementary_Functions G.1.2(6)
in Ada.Numerics.Generic_Elementary_Functions A.5.1(7)
Cot
in Ada.Numerics.Generic_Complex_Elementary_Functions G.1.2(4)
in Ada.Numerics.Generic_Elementary_Functions A.5.1(5)
Coth
in Ada.Numerics.Generic_Complex_Elementary_Functions G.1.2(6)
in Ada.Numerics.Generic_Elementary_Functions A.5.1(7)
Count
in Ada.Strings.Bounded A.4.4(48), A.4.4(49), A.4.4(50)
in Ada.Strings.Fixed A.4.3(13), A.4.3(14), A.4.3(15)
in Ada.Strings.Unbounded A.4.5(43), A.4.5(44), A.4.5(45)
Country *in* Ada.Locales A.19(6/3)
Create
in Ada.Direct_IO A.8.4(6), A.8.4(18.10/5), A.8.4(18.3/5)
in Ada.Sequential_IO A.8.1(6), A.8.1(15.10/5), A.8.1(15.3/5)
in Ada.Streams.Stream_IO A.12.1(8), A.12.1(26.10/5), A.12.1(26.3/5)
in Ada.Text_IO A.10.1(9), A.10.1(85.10/5), A.10.1(85.3/5)
in System.Multiprocessors.Dispatching_Domains D.16.1(7/4), D.16.1(9.2/4)
Create_Directory *in* Ada.Directories A.16(7/2)
Create_Path *in* Ada.Directories A.16(9/2)
Create_Subpool
in System.Storage_Pools.Subpools 13.11.4(7/3)

Current_Directory *in* Ada.Directories A.16(5/2)
Current_Error *in* Ada.Text_IO A.10.1(17), A.10.1(20)
Current_Handler
  in Ada.Execution_Time.Group_Budgets D.14.2(10/2)
  in Ada.Execution_Time.Timers D.14.1(7/2)
  in Ada.Interrupts C.3.2(6)
  in Ada.Real_Time.Timing_Events D.15(5/2)
Current_Indent
  in Ada.Strings.Text_Buffers A.4.12(14/5)
Current_Input *in* Ada.Text_IO A.10.1(17), A.10.1(20)
Current_Output *in* Ada.Text_IO A.10.1(17), A.10.1(20)
Current_State
  in Ada.Synchronous_Task_Control D.10(4/5)
Current_Task
  in Ada.Task_Identification C.7.1(3/5)
Current_Task_Fallback_Handler
  in Ada.Task_Termination C.7.3(5/2)
Current_Use
  in Ada.Containers.Bounded_Priority_Queues A.18.31(7/5)
  in Ada.Containers.Bounded_Synchronized_Queues A.18.29(6/5)
  in Ada.Containers.Synchronized_Queue_Interfaces A.18.27(7/5)
  in Ada.Containers.Unbounded_Priority_Queues A.18.30(7/5)
  in Ada.Containers.Unbounded_Synchronized_Queues A.18.28(6/5)
Day
  in Ada.Calendar 9.6(13)
  in Ada.Calendar.Formatting 9.6.1(23/2)
Day_of_Week
  in Ada.Calendar.Formatting 9.6.1(18/2)
Deallocate
  in System.Storage_Pools 13.11(8)
  in System.Storage_Pools.Subpools 13.11.4(15/3)
Deallocate_Subpool
  in System.Storage_Pools.Subpools 13.11.4(12/3)
Decode
  in Ada.Strings.UTF_Encoding.Strings A.4.11(26/3), A.4.11(27/3), A.4.11(28/3)
  in Ada.Strings.UTF_Encoding.Wide_Strings A.4.11(34/3), A.4.11(35/3), A.4.11(36/3)
  in Ada.Strings.UTF_Encoding.Wide_Wide_Strings A.4.11(42/3), A.4.11(43/3), A.4.11(44/3)
Decrease_Indent
  in Ada.Strings.Text_Buffers A.4.12(16/5)
Decrement *in* Interfaces.C.Pointers B.3.2(11/3)
Default_Modulus
  in Ada.Containers.Bounded_Hashed_Maps A.18.21(10/3)
  in Ada.Containers.Bounded_Hashed_Sets A.18.23(10/3)
Default_Subpool_for_Pool
  in System.Storage_Pools.Subpools 13.11.4(13/3)
Delay_Until_And_Set_CPU
  in System.Multiprocessors.Dispatching_Domains D.16.1(14/3)
Delay_Until_And_Set_Deadline
  in Ada.Dispatching.EDF D.2.6(9/5)
Delete
  in Ada.Containers.Doubly_Linked_Lists A.18.3(24/5)
  in Ada.Containers.Hashed_Maps A.18.5(25/5), A.18.5(26/5)
  in Ada.Containers.Hashed_Sets A.18.8(24/5), A.18.8(25/5), A.18.8(55/5)
  in Ada.Containers.Ordered_Maps A.18.6(24/5), A.18.6(25/5)
  in Ada.Containers.Ordered_Sets A.18.9(23/5), A.18.9(24/5), A.18.9(68/5)
  in Ada.Containers.Vectors A.18.2(50/5), A.18.2(51/5)
  in Ada.Direct_IO A.8.4(8)
  in Ada.Sequential_IO A.8.1(8)
  in Ada.Streams.Stream_IO A.12.1(10)
  in Ada.Strings.Bounded A.4.4(64), A.4.4(65)
  in Ada.Strings.Fixed A.4.3(29), A.4.3(30)
  in Ada.Strings.Unbounded A.4.5(59), A.4.5(60)
  in Ada.Text_IO A.10.1(11)
Delete_Children
  in Ada.Containers.Multiway_Trees A.18.10(53/5)
Delete_Directory *in* Ada.Directories A.16(8/2)
Delete_File *in* Ada.Directories A.16(11/2)
Delete_First
  in Ada.Containers.Doubly_Linked_Lists A.18.3(25/5)
  in Ada.Containers.Ordered_Maps A.18.6(26/5)
  in Ada.Containers.Ordered_Sets A.18.9(25/5)
  in Ada.Containers.Vectors A.18.2(52/5)
Delete_Last
  in Ada.Containers.Doubly_Linked_Lists A.18.3(26/5)
  in Ada.Containers.Ordered_Maps A.18.6(27/5)
  in Ada.Containers.Ordered_Sets A.18.9(26/5)
  in Ada.Containers.Vectors A.18.2(53/5)
Delete_Leaf
  in Ada.Containers.Multiway_Trees A.18.10(35/5)
Delete_Subtree
  in Ada.Containers.Multiway_Trees A.18.10(36/5)
Delete_Tree *in* Ada.Directories A.16(10/2)
Denominator
  in Ada.Numerics.Big_Numbers.Big_Reals A.5.7(8/5)
Depth
  in Ada.Containers.Multiway_Trees A.18.10(19.1/5), A.18.10(19/5)
Dequeue
  in Ada.Containers.Bounded_Priority_Queues A.18.31(5/3)
  in Ada.Containers.Bounded_Synchronized_Queues A.18.29(5/3)
  in Ada.Containers.Synchronized_Queue_Interfaces A.18.27(6/5)
  in Ada.Containers.Unbounded_Priority_Queues A.18.30(5/3)
  in Ada.Containers.Unbounded_Synchronized_Queues A.18.28(5/3)
Dequeue_Only_High_Priority
  in Ada.Containers.Bounded_Priority_Queues A.18.31(6/3)
  in Ada.Containers.Unbounded_Priority_Queues A.18.30(6/3)
Dereference_Error
  in Interfaces.C.Strings B.3.1(12)
Descendant_Tag *in* Ada.Tags 3.9(7.1/2)
Detach_Handler *in* Ada.Interrupts C.3.2(9)
Determinant
  in Ada.Numerics.Generic_Complex_Arrays G.3.2(46/2)
  in Ada.Numerics.Generic_Real_Arrays G.3.1(24/2)
Difference
  in Ada.Calendar.Arithmetic 9.6.1(12/2)
  in Ada.Containers.Hashed_Sets A.18.8(32/5), A.18.8(33/5)
  in Ada.Containers.Ordered_Sets A.18.9(33/5), A.18.9(34/5)
Divide *in* Ada.Decimal F.2(6/3)
Do_APC *in* System.RPC E.5(10)
Do_RPC *in* System.RPC E.5(9)
Eigensystem
  in Ada.Numerics.Generic_Complex_Arrays G.3.2(49/2)
  in Ada.Numerics.Generic_Real_Arrays G.3.1(27/2)
Eigenvalues
  in Ada.Numerics.Generic_Complex_Arrays G.3.2(48/2)
  in Ada.Numerics.Generic_Real_Arrays G.3.1(26/2)

Element
  in Ada.Containers.Doubly_Linked_Lists A.18.3(14.1/5), A.18.3(14/5)
  in Ada.Containers.Hashed_Maps A.18.5(14.1/5), A.18.5(14/5), A.18.5(31/2)
  in Ada.Containers.Hashed_Sets A.18.8(15.1/5), A.18.8(15/5), A.18.8(52/2)
  in Ada.Containers.Indefinite_Holders A.18.18(12/5)
  in Ada.Containers.Multiway_Trees A.18.10(24.1/5), A.18.10(24/5)
  in Ada.Containers.Ordered_Maps A.18.6(13.1/5), A.18.6(13/5), A.18.6(39/2)
  in Ada.Containers.Ordered_Sets A.18.9(14.1/5), A.18.9(14/5), A.18.9(65/2)
  in Ada.Containers.Vectors A.18.2(27/5), A.18.2(28.1/5), A.18.2(28/5)
  in Ada.Strings.Bounded A.4.4(26)
  in Ada.Strings.Unbounded A.4.5(20)
Element_Count
  in Ada.Streams.Storage 13.13.1(14/5)
  in Ada.Streams.Storage.Bounded 13.13.1(30/5)
  in Ada.Streams.Storage.Unbounded 13.13.1(22/5)
Empty
  in Ada.Containers.Doubly_Linked_Lists A.18.3(10.3/5)
  in Ada.Containers.Hashed_Maps A.18.5(7.3/5)
  in Ada.Containers.Hashed_Sets A.18.8(8.2/5)
  in Ada.Containers.Indefinite_Holders A.18.18(8.2/5)
  in Ada.Containers.Multiway_Trees A.18.10(15.3/5)
  in Ada.Containers.Ordered_Maps A.18.6(8.3/5)
  in Ada.Containers.Ordered_Sets A.18.9(9.2/5)
  in Ada.Containers.Vectors A.18.2(12.4/5)
Encode
  in Ada.Strings.UTF_Encoding.Strings A.4.11(23/3), A.4.11(24/3), A.4.11(25/3)
  in Ada.Strings.UTF_Encoding.Wide_Strings A.4.11(31/3), A.4.11(32/3), A.4.11(33/3)
  in Ada.Strings.UTF_Encoding.Wide_Wide_Strings A.4.11(39/3), A.4.11(40/3), A.4.11(41/3)
Encoding *in* Ada.Strings.UTF_Encoding A.4.11(13/3)
End_Of_File
  in Ada.Direct_IO A.8.4(16)
  in Ada.Sequential_IO A.8.1(13)
  in Ada.Streams.Stream_IO A.12.1(12)
  in Ada.Text_IO A.10.1(34/5)
End_Of_Line *in* Ada.Text_IO A.10.1(30/5)
End_Of_Page *in* Ada.Text_IO A.10.1(33/5)
End_Search *in* Ada.Directories A.16(33/2)
Enqueue
  in Ada.Containers.Bounded_Priority_Queues A.18.31(5/3)
  in Ada.Containers.Bounded_Synchronized_Queues A.18.29(5/3)
  in Ada.Containers.Synchronized_Queue_Interfaces A.18.27(5/5)
  in Ada.Containers.Unbounded_Priority_Queues A.18.30(5/3)
  in Ada.Containers.Unbounded_Synchronized_Queues A.18.28(5/3)
Environment_Task
  in Ada.Task_Identification C.7.1(3/5)
Equal_Case_Insensitive
  child of Ada.Strings A.4.10(2/5)
  child of Ada.Strings.Bounded A.4.10(7/5)
  child of Ada.Strings.Fixed A.4.10(5/3)
  child of Ada.Strings.Unbounded A.4.10(10/5)
Equal_Element
  in Ada.Containers.Indefinite_Holders A.18.18(7.1/5)
  in Ada.Containers.Multiway_Trees A.18.10(11.1/5)
Equal_Subtree
  in Ada.Containers.Multiway_Trees A.18.10(14/3)
Equivalent_Elements
  in Ada.Containers.Hashed_Sets A.18.8(46/5), A.18.8(47/5), A.18.8(48/5)
  in Ada.Containers.Ordered_Sets A.18.9(3/2)
Equivalent_Keys
  in Ada.Containers.Hashed_Maps A.18.5(34/5), A.18.5(35/5), A.18.5(36/5)
  in Ada.Containers.Ordered_Maps A.18.6(3/5)
  in Ada.Containers.Ordered_Sets A.18.9(63/2)
Equivalent_Sets
  in Ada.Containers.Hashed_Sets A.18.8(8/2)
  in Ada.Containers.Ordered_Sets A.18.9(9/2)
Establish_RPC_Receiver *in* System.RPC E.5(12)
Exception_Identity *in* Ada.Exceptions 11.4.1(5/2)
Exception_Information
  in Ada.Exceptions 11.4.1(5/2)
Exception_Message *in* Ada.Exceptions 11.4.1(4/3)
Exception_Name *in* Ada.Exceptions 11.4.1(2/5), 11.4.1(5/2)
Exchange_Handler *in* Ada.Interrupts C.3.2(8)
Exclude
  in Ada.Containers.Hashed_Maps A.18.5(24/5)
  in Ada.Containers.Hashed_Sets A.18.8(23/5), A.18.8(54/5)
  in Ada.Containers.Ordered_Maps A.18.6(23/5)
  in Ada.Containers.Ordered_Sets A.18.9(22/5), A.18.9(67/5)
Exists
  in Ada.Directories A.16(24/2)
  in Ada.Environment_Variables A.17(5/2)
Exp
  in Ada.Numerics.Generic_Complex_Elementary_Functions G.1.2(3)
  in Ada.Numerics.Generic_Elementary_Functions A.5.1(4)
Expanded_Name *in* Ada.Tags 3.9(7/2)
Extension *in* Ada.Directories A.16(18/5)
External_Tag *in* Ada.Tags 3.9(7/2)
Finalize *in* Ada.Finalization 7.6(6/2), 7.6(8/2)
Find
  in Ada.Containers.Doubly_Linked_Lists A.18.3(41/5)
  in Ada.Containers.Hashed_Maps A.18.5(30/5)
  in Ada.Containers.Hashed_Sets A.18.8(43/5), A.18.8(56/5)
  in Ada.Containers.Multiway_Trees A.18.10(38/5)
  in Ada.Containers.Ordered_Maps A.18.6(38/5)
  in Ada.Containers.Ordered_Sets A.18.9(49/5), A.18.9(69/5)
  in Ada.Containers.Vectors A.18.2(68/5)
Find_In_Subtree
  in Ada.Containers.Multiway_Trees A.18.10(39.1/5), A.18.10(39/5)
Find_Index *in* Ada.Containers.Vectors A.18.2(67/2)
Find_Token
  in Ada.Strings.Bounded A.4.4(50.1/3), A.4.4(51)
  in Ada.Strings.Fixed A.4.3(15.1/3), A.4.3(16)
  in Ada.Strings.Unbounded A.4.5(45.1/3), A.4.5(46)
First
  in Ada.Containers.Doubly_Linked_Lists A.18.3(33/5)
  in Ada.Containers.Hashed_Maps A.18.5(27/5)
  in Ada.Containers.Hashed_Sets A.18.8(40/5)
  in Ada.Containers.Ordered_Maps A.18.6(28/5)
  in Ada.Containers.Ordered_Sets A.18.9(41/5)
  in Ada.Containers.Vectors A.18.2(58/5)
  in Ada.Iterator_Interfaces 5.5.1(3/3), 5.5.1(4.6/5)

First_Child
in Ada.Containers.Multiway_Trees A.18.10(60.1/5), A.18.10(60/5)
First_Child_Element
in Ada.Containers.Multiway_Trees A.18.10(61.1/5), A.18.10(61/5)
First_Element
in Ada.Containers.Doubly_Linked_Lists A.18.3(34/5)
in Ada.Containers.Ordered_Maps A.18.6(29/5)
in Ada.Containers.Ordered_Sets A.18.9(42/5)
in Ada.Containers.Vectors A.18.2(59/5)
First_Index *in* Ada.Containers.Vectors A.18.2(57/5)
First_Key
in Ada.Containers.Ordered_Maps A.18.6(30/5)
Floor
in Ada.Containers.Ordered_Maps A.18.6(40/5)
in Ada.Containers.Ordered_Sets A.18.9(50/2), A.18.9(70/5)
Flush
in Ada.Direct_IO A.8.4(10.1/5)
in Ada.Sequential_IO A.8.1(10.1/5)
in Ada.Streams.Stream_IO A.12.1(25/1)
in Ada.Text_IO A.10.1(21/5)
Form
in Ada.Direct_IO A.8.4(9), A.8.4(18.13/5), A.8.4(18.6/5)
in Ada.Sequential_IO A.8.1(9), A.8.1(15.13/5), A.8.1(15.6/5)
in Ada.Streams.Stream_IO A.12.1(11), A.12.1(26.13/5), A.12.1(26.6/5)
in Ada.Text_IO A.10.1(12), A.10.1(85.13/5), A.10.1(85.6/5)
Free
in Ada.Strings.Unbounded A.4.5(7)
in Interfaces.C.Strings B.3.1(11)
From_Big_Integer
in Ada.Numerics.Big_Numbers.Big_Integers A.5.6(12/5), A.5.6(13/5)
From_Big_Real
in Ada.Numerics.Big_Numbers.Big_Reals A.5.7(13/5), A.5.7(14/5)
From_Quotient_String
in Ada.Numerics.Big_Numbers.Big_Reals A.5.7(19/5)
From_String
in Ada.Numerics.Big_Numbers.Big_Integers A.5.6(15/5)
in Ada.Numerics.Big_Numbers.Big_Reals A.5.7(16/5)
From_Universal_Image
in Ada.Numerics.Big_Numbers.Big_Integers A.5.6(16/5)
in Ada.Numerics.Big_Numbers.Big_Reals A.5.7(17/5), A.5.7(18/5)
Full_Name *in* Ada.Directories A.16(15/5), A.16(39/2)
Generic_Array_Sort
child of Ada.Containers A.18.26(3/5)
Generic_Constrained_Array_Sort
child of Ada.Containers A.18.26(7/5)
Generic_Sort
child of Ada.Containers A.18.26(9.2/5)
Get
in Ada.Strings.Text_Buffers.Unbounded A.4.12(20/5)
in Ada.Text_IO A.10.1(41/5), A.10.1(47/5), A.10.1(54/5), A.10.1(55/5), A.10.1(59/5), A.10.1(60/5), A.10.1(65/5), A.10.1(67/5), A.10.1(70/5), A.10.1(72/5), A.10.1(75/5), A.10.1(77/5), A.10.1(81/5), A.10.1(83/5)
in Ada.Text_IO.Complex_IO G.1.3(6), G.1.3(8/5)
Get_CPU
in Ada.Interrupts C.3.2(10.1/3)
in System.Multiprocessors.Dispatching_Domains D.16.1(13/3)
Get_CPU_Set
in System.Multiprocessors.Dispatching_Domains D.16.1(9.3/4)
Get_Deadline *in* Ada.Dispatching.EDF D.2.6(9/5)
Get_Dispatching_Domain
in System.Multiprocessors.Dispatching_Domains D.16.1(10/3)
Get_First_CPU
in System.Multiprocessors.Dispatching_Domains D.16.1(8/3)
Get_Immediate *in* Ada.Text_IO A.10.1(44/5), A.10.1(45/5)
Get_Last_CPU
in System.Multiprocessors.Dispatching_Domains D.16.1(9/4)
Get_Last_Release_Time
in Ada.Dispatching.EDF D.2.6(9/5)
Get_Line
in Ada.Text_IO A.10.1(49.1/5), A.10.1(49/5)
in Ada.Text_IO.Bounded_IO A.10.11(8/2), A.10.11(9/2), A.10.11(10/2), A.10.11(11/2)
in Ada.Text_IO.Unbounded_IO A.10.12(8/2), A.10.12(9/2), A.10.12(10/2), A.10.12(11/2)
Get_Next_Entry *in* Ada.Directories A.16(35/2)
Get_Priority
in Ada.Dynamic_Priorities D.5.1(5)
Get_Relative_Deadline
in Ada.Dispatching.EDF D.2.6(9/5)
Get_UTF_8
in Ada.Strings.Text_Buffers.Unbounded A.4.12(23/5)
Greatest_Common_Divisor
in Ada.Numerics.Big_Numbers.Big_Integers A.5.6(19/5)
Has_Element
in Ada.Containers.Doubly_Linked_Lists A.18.3(9.1/5), A.18.3(9.2/5), A.18.3(51.6/5), A.18.3(69.4/5)
in Ada.Containers.Hashed_Maps A.18.5(6.1/5), A.18.5(6.2/5), A.18.5(37.7/5)
in Ada.Containers.Hashed_Sets A.18.8(6.1/5), A.18.8(6.2/5), A.18.8(59.6/5)
in Ada.Containers.Multiway_Trees A.18.10(12.1/5), A.18.10(12/5), A.18.10(70.6/5), A.18.10(92.1/5)
in Ada.Containers.Ordered_Maps A.18.6(7.1/5), A.18.6(7.2/5), A.18.6(51.8/5)
in Ada.Containers.Ordered_Sets A.18.9(7.1/5), A.18.9(7.2/5), A.18.9(74.6/5)
in Ada.Containers.Vectors A.18.2(11.1/5), A.18.2(11.2/5), A.18.2(79.6/5), A.18.2(97.4/5)
Hash
child of Ada.Strings A.4.9(2/5)
child of Ada.Strings.Bounded A.4.9(7/5)
child of Ada.Strings.Unbounded A.4.9(10/5)
Hash_Case_Insensitive
child of Ada.Strings A.4.9(11.2/5)
child of Ada.Strings.Bounded A.4.9(11.7/5)
child of Ada.Strings.Fixed A.4.9(11.5/3)
child of Ada.Strings.Unbounded A.4.9(11.10/5)
Head
in Ada.Strings.Bounded A.4.4(70), A.4.4(71)
in Ada.Strings.Fixed A.4.3(35), A.4.3(36)
in Ada.Strings.Unbounded A.4.5(65), A.4.5(66)
Hold *in* Ada.Asynchronous_Task_Control D.11(3/5)
Hour *in* Ada.Calendar.Formatting 9.6.1(24/2)

Im
  in Ada.Numerics.Generic_Complex_Arrays G.3.2(7/2), G.3.2(27/2)
  in Ada.Numerics.Generic_Complex_Types G.1.1(6)
Image
  in Ada.Calendar.Formatting 9.6.1(35/2), 9.6.1(37/2)
  in Ada.Numerics.Discrete_Random A.5.2(26)
  in Ada.Numerics.Float_Random A.5.2(14)
  in Ada.Task_Identification C.7.1(3/5)
  in Ada.Text_IO.Editing F.3.3(13)
In_Range
  in Ada.Numerics.Big_Numbers.Big_Integers A.5.6(10/5)
  in Ada.Numerics.Big_Numbers.Big_Reals A.5.7(12/5)
Include
  in Ada.Containers.Hashed_Maps A.18.5(22/5)
  in Ada.Containers.Hashed_Sets A.18.8(21/5)
  in Ada.Containers.Ordered_Maps A.18.6(21/5)
  in Ada.Containers.Ordered_Sets A.18.9(20/5)
Increase_Indent
  in Ada.Strings.Text_Buffers A.4.12(15/5)
Increment *in* Interfaces.C.Pointers B.3.2(11/3)
Index
  in Ada.Direct_IO A.8.4(15)
  in Ada.Streams.Stream_IO A.12.1(23)
  in Ada.Strings.Bounded A.4.4(43.1/2), A.4.4(43.2/2), A.4.4(44), A.4.4(45), A.4.4(45.1/2), A.4.4(46)
  in Ada.Strings.Fixed A.4.3(8.1/2), A.4.3(8.2/2), A.4.3(9), A.4.3(10), A.4.3(10.1/2), A.4.3(11)
  in Ada.Strings.Unbounded A.4.5(38.1/2), A.4.5(38.2/2), A.4.5(39), A.4.5(40), A.4.5(40.1/2), A.4.5(41)
Index_Non_Blank
  in Ada.Strings.Bounded A.4.4(46.1/2), A.4.4(47)
  in Ada.Strings.Fixed A.4.3(11.1/2), A.4.3(12)
  in Ada.Strings.Unbounded A.4.5(41.1/2), A.4.5(42)
Initial_Directory
  in Ada.Directories.Hierarchical_File_Names A.16.1(12/3)
Initialize *in* Ada.Finalization 7.6(6/2), 7.6(8/2)
Insert
  in Ada.Containers.Doubly_Linked_Lists A.18.3(19/5), A.18.3(20/5), A.18.3(21/5)
  in Ada.Containers.Hashed_Maps A.18.5(19/5), A.18.5(20/5), A.18.5(21/5)
  in Ada.Containers.Hashed_Sets A.18.8(19/5), A.18.8(20/5)
  in Ada.Containers.Ordered_Maps A.18.6(18/5), A.18.6(19/5), A.18.6(20/5)
  in Ada.Containers.Ordered_Sets A.18.9(18/5), A.18.9(19/5)
  in Ada.Containers.Vectors A.18.2(39/5), A.18.2(40/5), A.18.2(41/5), A.18.2(42/5), A.18.2(43/5)
  in Ada.Strings.Bounded A.4.4(60), A.4.4(61)
  in Ada.Strings.Fixed A.4.3(25), A.4.3(26)
  in Ada.Strings.Unbounded A.4.5(55), A.4.5(56)
Insert_Child
  in Ada.Containers.Multiway_Trees A.18.10(48/5), A.18.10(49/5), A.18.10(50/5)
Insert_Space
  in Ada.Containers.Vectors A.18.2(48/5), A.18.2(49/5)
Insert_Vector
  in Ada.Containers.Vectors A.18.2(36/5), A.18.2(37/5), A.18.2(38/5)
Interface_Ancestor_Tags *in* Ada.Tags 3.9(7.4/2)
Internal_Tag *in* Ada.Tags 3.9(7/2)
Intersection
  in Ada.Containers.Hashed_Sets A.18.8(29/5), A.18.8(30/5)
  in Ada.Containers.Ordered_Sets A.18.9(30/5), A.18.9(31/5)
Inverse
  in Ada.Numerics.Generic_Complex_Arrays G.3.2(46/2)
  in Ada.Numerics.Generic_Real_Arrays G.3.1(24/2)
Is_A_Group_Member
  in Ada.Execution_Time.Group_Budgets D.14.2(8/2)
Is_Abstract *in* Ada.Tags 3.9(7.5/3)
Is_Alphanumeric
  in Ada.Characters.Handling A.3.2(4/5)
  in Ada.Wide_Characters.Handling A.3.5(12/3)
Is_Ancestor_Of
  in Ada.Containers.Multiway_Trees A.18.10(21.2/5)
Is_Attached *in* Ada.Interrupts C.3.2(5)
Is_Basic
  in Ada.Characters.Handling A.3.2(4/5)
  in Ada.Wide_Characters.Handling A.3.5(8.1/5)
Is_Callable
  in Ada.Task_Identification C.7.1(4/3)
Is_Character
  in Ada.Characters.Conversions A.3.4(3/2)
Is_Control
  in Ada.Characters.Handling A.3.2(4/5)
  in Ada.Wide_Characters.Handling A.3.5(5/3)
Is_Current_Directory_Name
  in Ada.Directories.Hierarchical_File_Names A.16.1(7/3)
Is_Decimal_Digit
  in Ada.Characters.Handling A.3.2(4/5)
  in Ada.Wide_Characters.Handling A.3.5(10/3)
Is_Descendant_At_Same_Level
  in Ada.Tags 3.9(7.1/2)
Is_Digit
  in Ada.Characters.Handling A.3.2(4/5)
  in Ada.Wide_Characters.Handling A.3.5(9/3)
Is_Empty
  in Ada.Containers.Doubly_Linked_Lists A.18.3(12/5)
  in Ada.Containers.Hashed_Maps A.18.5(11/5)
  in Ada.Containers.Hashed_Sets A.18.8(13/5)
  in Ada.Containers.Indefinite_Holders A.18.18(10/5)
  in Ada.Containers.Multiway_Trees A.18.10(16/5)
  in Ada.Containers.Ordered_Maps A.18.6(10/5)
  in Ada.Containers.Ordered_Sets A.18.9(12/5)
  in Ada.Containers.Vectors A.18.2(23/5)
Is_Full_Name
  in Ada.Directories.Hierarchical_File_Names A.16.1(8/3)
Is_Graphic
  in Ada.Characters.Handling A.3.2(4/5)
  in Ada.Wide_Characters.Handling A.3.5(19/3)
Is_Held
  in Ada.Asynchronous_Task_Control D.11(3/5)
Is_Hexadecimal_Digit
  in Ada.Characters.Handling A.3.2(4/5)
  in Ada.Wide_Characters.Handling A.3.5(11/3)
Is_In
  in Ada.Strings.Maps A.4.2(13)
  in Ada.Strings.Wide_Maps A.4.7(13)
  in Ada.Strings.Wide_Wide_Maps A.4.8(13/2)
Is_ISO_646 *in* Ada.Characters.Handling A.3.2(10)
Is_Leaf
  in Ada.Containers.Multiway_Trees A.18.10(21.1/5), A.18.10(21/5)
Is_Letter
  in Ada.Characters.Handling A.3.2(4/5)
  in Ada.Wide_Characters.Handling A.3.5(6/3)

Is_Line_Terminator
in Ada.Characters.Handling A.3.2(4/5)
in Ada.Wide_Characters.Handling A.3.5(14/3)
Is_Lock_Free
in System.Atomic_Operations.Exchange C.6.2(8/5)
in System.Atomic_Operations.Integer_Arithmetic C.6.4(8/5)
in System.Atomic_Operations.Modular_Arithmetic C.6.5(8/5)
in System.Atomic_Operations.Test_And_Set C.6.3(7/5)
Is_Lower
in Ada.Characters.Handling A.3.2(4/5)
in Ada.Wide_Characters.Handling A.3.5(7/3)
Is_Mark
in Ada.Characters.Handling A.3.2(4/5)
in Ada.Wide_Characters.Handling A.3.5(15/3)
Is_Member
in Ada.Execution_Time.Group_Budgets D.14.2(8/2)
Is_NFKC
in Ada.Characters.Handling A.3.2(4/5)
in Ada.Wide_Characters.Handling A.3.5(18.1/5)
Is_Nul_Terminated *in* Interfaces.C B.3(24), B.3(35), B.3(39.16/2), B.3(39.7/2)
Is_Open
in Ada.Direct_IO A.8.4(10)
in Ada.Sequential_IO A.8.1(10)
in Ada.Streams.Stream_IO A.12.1(12)
in Ada.Text_IO A.10.1(13)
Is_Other_Format
in Ada.Characters.Handling A.3.2(4/5)
in Ada.Wide_Characters.Handling A.3.5(16/3)
Is_Parent_Directory_Name
in Ada.Directories.Hierarchical_File_Names A.16.1(6/3)
Is_Punctuation_Connector
in Ada.Characters.Handling A.3.2(4/5)
in Ada.Wide_Characters.Handling A.3.5(17/3)
Is_Relative_Name
in Ada.Directories.Hierarchical_File_Names A.16.1(9/3)
Is_Reserved *in* Ada.Interrupts C.3.2(4)
Is_Root
in Ada.Containers.Multiway_Trees A.18.10(20.1/5), A.18.10(20/5)
Is_Root_Directory_Name
in Ada.Directories.Hierarchical_File_Names A.16.1(5/3)
Is_Round_Robin
in Ada.Dispatching.Round_Robin D.2.5(4/5)
Is_Simple_Name
in Ada.Directories.Hierarchical_File_Names A.16.1(4/3)
Is_Sorted
in Ada.Containers.Doubly_Linked_Lists A.18.3(48/2)
in Ada.Containers.Vectors A.18.2(76/2)
Is_Space
in Ada.Characters.Handling A.3.2(4/5)
in Ada.Wide_Characters.Handling A.3.5(18/3)
Is_Special
in Ada.Characters.Handling A.3.2(4/5)
in Ada.Wide_Characters.Handling A.3.5(13/3)
Is_Split *in* Ada.Iterator_Interfaces 5.5.1(4.3/5)
Is_String
in Ada.Characters.Conversions A.3.4(3/2)
Is_Subset
in Ada.Containers.Hashed_Sets A.18.8(39/2)
in Ada.Containers.Ordered_Sets A.18.9(40/2)
in Ada.Strings.Maps A.4.2(14)
in Ada.Strings.Wide_Maps A.4.7(14)
in Ada.Strings.Wide_Wide_Maps A.4.8(14/2)
Is_Terminated
in Ada.Task_Identification C.7.1(4/3)
Is_Upper
in Ada.Characters.Handling A.3.2(4/5)
in Ada.Wide_Characters.Handling A.3.5(8/3)
Is_Valid
in Ada.Numerics.Big_Numbers.Big_Integers A.5.6(4/5)
in Ada.Numerics.Big_Numbers.Big_Reals A.5.7(4/5)
Is_Wide_Character
in Ada.Characters.Conversions A.3.4(3/2)
Is_Wide_String
in Ada.Characters.Conversions A.3.4(3/2)
Iterate
in Ada.Containers.Doubly_Linked_Lists A.18.3(45/5), A.18.3(46.1/5), A.18.3(46.2/5)
in Ada.Containers.Hashed_Maps A.18.5(37/5)
in Ada.Containers.Hashed_Sets A.18.8(49/5)
in Ada.Containers.Multiway_Trees A.18.10(42/5), A.18.10(44/5)
in Ada.Containers.Ordered_Maps A.18.6(50/5), A.18.6(51.1/5), A.18.6(51.2/5)
in Ada.Containers.Ordered_Sets A.18.9(60/5)
in Ada.Containers.Vectors A.18.2(73/5), A.18.2(74.1/5), A.18.2(74.2/5)
in Ada.Environment_Variables A.17(8/5)
Iterate_Children
in Ada.Containers.Multiway_Trees A.18.10(68.1/5), A.18.10(68/5), A.18.10(70/5)
Iterate_Subtree
in Ada.Containers.Multiway_Trees A.18.10(43.1/5), A.18.10(43/5), A.18.10(45.1/5), A.18.10(45/5)
Key
in Ada.Containers.Hashed_Maps A.18.5(13.1/5), A.18.5(13/5)
in Ada.Containers.Hashed_Sets A.18.8(51.1/5), A.18.8(51/5)
in Ada.Containers.Ordered_Maps A.18.6(12.1/5), A.18.6(12/5)
in Ada.Containers.Ordered_Sets A.18.9(64.1/5), A.18.9(64/5)
Kind *in* Ada.Directories A.16(25/2), A.16(40/2)
Language *in* Ada.Locales A.19(6/3)
Last
in Ada.Containers.Doubly_Linked_Lists A.18.3(35/5)
in Ada.Containers.Ordered_Maps A.18.6(31/5)
in Ada.Containers.Ordered_Sets A.18.9(43/5)
in Ada.Containers.Vectors A.18.2(61/5)
in Ada.Iterator_Interfaces 5.5.1(4/3)
Last_Child
in Ada.Containers.Multiway_Trees A.18.10(62.1/5), A.18.10(62/5)
Last_Child_Element
in Ada.Containers.Multiway_Trees A.18.10(63.1/5), A.18.10(63/5)
Last_Element
in Ada.Containers.Doubly_Linked_Lists A.18.3(36/5)
in Ada.Containers.Ordered_Maps A.18.6(32/5)
in Ada.Containers.Ordered_Sets A.18.9(44/5)
in Ada.Containers.Vectors A.18.2(62/5)
Last_Index *in* Ada.Containers.Vectors A.18.2(60/5)
Last_Key
in Ada.Containers.Ordered_Maps A.18.6(33/5)

Length
in Ada.Containers.Doubly_Linked_Lists A.18.3(11/5)
in Ada.Containers.Hashed_Maps A.18.5(10/5)
in Ada.Containers.Hashed_Sets A.18.8(12/5)
in Ada.Containers.Ordered_Maps A.18.6(9/5)
in Ada.Containers.Ordered_Sets A.18.9(11/5)
in Ada.Containers.Vectors A.18.2(21/5)
in Ada.Strings.Bounded A.4.4(9)
in Ada.Strings.Unbounded A.4.5(6)
in Ada.Text_IO.Editing F.3.3(11)
in Interfaces.COBOL B.4(34), B.4(39), B.4(44)
Less_Case_Insensitive
child of Ada.Strings A.4.10(13/5)
child of Ada.Strings.Bounded A.4.10(18/5)
child of Ada.Strings.Fixed A.4.10(16/3)
child of Ada.Strings.Unbounded A.4.10(21/5)
Line *in* Ada.Text_IO A.10.1(38/5)
Line_Length *in* Ada.Text_IO A.10.1(25/5)
Local_Image
in Ada.Calendar.Formatting 9.6.1(35.1/5)
Local_Time_Offset
in Ada.Calendar.Time_Zones 9.6.1(6/5)
Log
in Ada.Numerics.Generic_Complex_Elementary_Functions G.1.2(3)
in Ada.Numerics.Generic_Elementary_Functions A.5.1(4)
Look_Ahead *in* Ada.Text_IO A.10.1(43/5)
Max
in Ada.Numerics.Big_Numbers.Big_Integers A.5.6(18/5)
in Ada.Numerics.Big_Numbers.Big_Reals A.5.7(21/5)
Maximum_Length
in Ada.Containers.Vectors A.18.2(12.3/5)
Meaningful_For
in Ada.Containers.Multiway_Trees A.18.10(22.1/5)
Members
in Ada.Execution_Time.Group_Budgets D.14.2(8/2)
Merge
in Ada.Containers.Doubly_Linked_Lists A.18.3(50/5)
in Ada.Containers.Vectors A.18.2(78/5)
Microseconds *in* Ada.Real_Time D.8(14/2)
Milliseconds *in* Ada.Real_Time D.8(14/2)
Min
in Ada.Numerics.Big_Numbers.Big_Integers A.5.6(18/5)
in Ada.Numerics.Big_Numbers.Big_Reals A.5.7(21/5)
Minute *in* Ada.Calendar.Formatting 9.6.1(25/2)
Minutes *in* Ada.Real_Time D.8(14/2)
Mode
in Ada.Direct_IO A.8.4(9)
in Ada.Sequential_IO A.8.1(9)
in Ada.Streams.Stream_IO A.12.1(11)
in Ada.Text_IO A.10.1(12)
Modification_Time *in* Ada.Directories A.16(27/2), A.16(42/2)
Modulus
in Ada.Numerics.Generic_Complex_Arrays G.3.2(10/2), G.3.2(30/2)
in Ada.Numerics.Generic_Complex_Types G.1.1(9)
Month
in Ada.Calendar 9.6(13)
in Ada.Calendar.Formatting 9.6.1(22/2)
More_Entries *in* Ada.Directories A.16(34/2)
Move
in Ada.Containers.Doubly_Linked_Lists A.18.3(18/5)
in Ada.Containers.Hashed_Maps A.18.5(18/5)
in Ada.Containers.Hashed_Sets A.18.8(18/5)
in Ada.Containers.Indefinite_Holders A.18.18(22/5)
in Ada.Containers.Multiway_Trees A.18.10(34/5)
in Ada.Containers.Ordered_Maps A.18.6(17/5)
in Ada.Containers.Ordered_Sets A.18.9(17/5)
in Ada.Containers.Vectors A.18.2(35/5)
in Ada.Strings.Fixed A.4.3(7)
Name
in Ada.Direct_IO A.8.4(9), A.8.4(18.12/5), A.8.4(18.5/5)
in Ada.Sequential_IO A.8.1(9), A.8.1(15.12/5), A.8.1(15.5/5)
in Ada.Streams.Stream_IO A.12.1(11), A.12.1(26.12/5), A.12.1(26.5/5)
in Ada.Text_IO A.10.1(12), A.10.1(85.12/5), A.10.1(85.5/5)
Name_Case_Equivalence
in Ada.Directories A.16(20.2/3)
Nanoseconds *in* Ada.Real_Time D.8(14/2)
New_Char_Array
in Interfaces.C.Strings B.3.1(9)
New_Line
in Ada.Strings.Text_Buffers A.4.12(12/5)
in Ada.Text_IO A.10.1(28/5)
New_Page *in* Ada.Text_IO A.10.1(31/5)
New_String *in* Interfaces.C.Strings B.3.1(10)
New_Vector *in* Ada.Containers.Vectors A.18.2(14.1/5)
Next
in Ada.Containers.Doubly_Linked_Lists A.18.3(37.1/5), A.18.3(37/5), A.18.3(39.1/5), A.18.3(39/5)
in Ada.Containers.Hashed_Maps A.18.5(28.1/5), A.18.5(28/5), A.18.5(29.1/5), A.18.5(29/5)
in Ada.Containers.Hashed_Sets A.18.8(41.1/5), A.18.8(41/5), A.18.8(42.1/5), A.18.8(42/5)
in Ada.Containers.Ordered_Maps A.18.6(34.1/5), A.18.6(34/5), A.18.6(35.1/5), A.18.6(35/5)
in Ada.Containers.Ordered_Sets A.18.9(45.1/5), A.18.9(45/5), A.18.9(46.1/5), A.18.9(46/5)
in Ada.Containers.Vectors A.18.2(63.1/5), A.18.2(63/5), A.18.2(64.1/5), A.18.2(64/5)
in Ada.Iterator_Interfaces 5.5.1(3/3), 5.5.1(4.7/5)
Next_Sibling
in Ada.Containers.Multiway_Trees A.18.10(64.1/5), A.18.10(64/5), A.18.10(65.1/5), A.18.10(65/5)
Node_Count
in Ada.Containers.Multiway_Trees A.18.10(17/5)
Null_Task_Id
in Ada.Task_Identification C.7.1(2/5)
Number_Of_CPUs
in System.Multiprocessors D.16(5/3)
Numerator
in Ada.Numerics.Big_Numbers.Big_Reals A.5.7(7/5)
Open
in Ada.Direct_IO A.8.4(7), A.8.4(18.11/5), A.8.4(18.4/5)
in Ada.Sequential_IO A.8.1(7), A.8.1(15.11/5), A.8.1(15.4/5)
in Ada.Streams.Stream_IO A.12.1(9), A.12.1(26.11/5), A.12.1(26.4/5)
in Ada.Text_IO A.10.1(10), A.10.1(85.11/5), A.10.1(85.4/5)
Overlap
in Ada.Containers.Hashed_Sets A.18.8(38/2)
in Ada.Containers.Ordered_Sets A.18.9(39/2)
Overwrite
in Ada.Strings.Bounded A.4.4(62), A.4.4(63)
in Ada.Strings.Fixed A.4.3(27), A.4.3(28)
in Ada.Strings.Unbounded A.4.5(57), A.4.5(58)
Page *in* Ada.Text_IO A.10.1(39/5)

Page_Length *in* Ada.Text_IO A.10.1(26/5)
Parent
in Ada.Containers.Multiway_Trees A.18.10(59.1/5), A.18.10(59/5)
Parent_Tag *in* Ada.Tags 3.9(7.2/2)
Peak_Use
in Ada.Containers.Bounded_Priority_Queues A.18.31(7/5)
in Ada.Containers.Bounded_Synchronized_Queues A.18.29(6/5)
in Ada.Containers.Synchronized_Queue_Interfaces A.18.27(7/5)
in Ada.Containers.Unbounded_Priority_Queues A.18.30(7/5)
in Ada.Containers.Unbounded_Synchronized_Queues A.18.28(6/5)
Pic_String *in* Ada.Text_IO.Editing F.3.3(7)
Pool_of_Subpool
in System.Storage_Pools.Subpools 13.11.4(9/3)
Prepend
in Ada.Containers.Doubly_Linked_Lists A.18.3(22/5)
in Ada.Containers.Vectors A.18.2(45/5)
Prepend_Child
in Ada.Containers.Multiway_Trees A.18.10(51/5)
Prepend_Vector
in Ada.Containers.Vectors A.18.2(44/5)
Previous
in Ada.Containers.Doubly_Linked_Lists A.18.3(38.1/5), A.18.3(38/5), A.18.3(40.1/5), A.18.3(40/5)
in Ada.Containers.Ordered_Maps A.18.6(36.1/5), A.18.6(36/5), A.18.6(37.1/5), A.18.6(37/5)
in Ada.Containers.Ordered_Sets A.18.9(47.1/5), A.18.9(47/5), A.18.9(48.1/5), A.18.9(48/5)
in Ada.Containers.Vectors A.18.2(65.1/5), A.18.2(65/5), A.18.2(66.1/5), A.18.2(66/5)
in Ada.Iterator_Interfaces 5.5.1(4/3)
Previous_Sibling
in Ada.Containers.Multiway_Trees A.18.10(66.1/5), A.18.10(66/5), A.18.10(67.1/5), A.18.10(67/5)
Put
in Ada.Strings.Text_Buffers A.4.12(7/5)
in Ada.Text_IO A.10.1(42/5), A.10.1(48/5), A.10.1(55/5), A.10.1(60/5), A.10.1(66/5), A.10.1(67/5), A.10.1(71/5), A.10.1(72/5), A.10.1(76/5), A.10.1(77/5), A.10.1(82/5), A.10.1(83/5)
in Ada.Text_IO.Bounded_IO A.10.11(4/2), A.10.11(5/2)
in Ada.Text_IO.Complex_IO G.1.3(7), G.1.3(8/5)
in Ada.Text_IO.Editing F.3.3(14/5), F.3.3(15/5), F.3.3(16)
in Ada.Text_IO.Unbounded_IO A.10.12(4/2), A.10.12(5/2)
Put_Image
in Ada.Numerics.Big_Numbers.Big_Integers A.5.6(17/5)
in Ada.Numerics.Big_Numbers.Big_Reals A.5.7(20/5)
Put_Line
in Ada.Text_IO A.10.1(50/5)
in Ada.Text_IO.Bounded_IO A.10.11(6/2), A.10.11(7/2)
in Ada.Text_IO.Unbounded_IO A.10.12(6/2), A.10.12(7/2)
Put_UTF_8 *in* Ada.Strings.Text_Buffers A.4.12(10/5)
Query_Element
in Ada.Containers.Doubly_Linked_Lists A.18.3(16.1/5), A.18.3(16/5)
in Ada.Containers.Hashed_Maps A.18.5(16.1/5), A.18.5(16/5)
in Ada.Containers.Hashed_Sets A.18.8(17.1/5), A.18.8(17/5)
in Ada.Containers.Indefinite_Holders A.18.18(14/5)
in Ada.Containers.Multiway_Trees A.18.10(26.1/5), A.18.10(26/5)
in Ada.Containers.Ordered_Maps A.18.6(15.1/5), A.18.6(15/5)
in Ada.Containers.Ordered_Sets A.18.9(16.1/5), A.18.9(16/5)
in Ada.Containers.Vectors A.18.2(31/5), A.18.2(32.1/5), A.18.2(32/5)
Raise_Exception *in* Ada.Exceptions 11.4.1(4/3)
Random
in Ada.Numerics.Discrete_Random A.5.2(20/5)
in Ada.Numerics.Float_Random A.5.2(8/5)
Re
in Ada.Numerics.Generic_Complex_Arrays G.3.2(7/2), G.3.2(27/2)
in Ada.Numerics.Generic_Complex_Types G.1.1(6)
Read
in Ada.Direct_IO A.8.4(12/5)
in Ada.Sequential_IO A.8.1(12/5)
in Ada.Storage_IO A.9(6)
in Ada.Streams 13.13.1(5)
in Ada.Streams.Storage.Bounded 13.13.1(28/5)
in Ada.Streams.Storage.Unbounded 13.13.1(20/5)
in Ada.Streams.Stream_IO A.12.1(15/5), A.12.1(16/5)
in System.RPC E.5(7)
Reference
in Ada.Containers.Doubly_Linked_Lists A.18.3(17.4/5)
in Ada.Containers.Hashed_Maps A.18.5(17.4/5), A.18.5(17.6/5)
in Ada.Containers.Indefinite_Holders A.18.18(19/5)
in Ada.Containers.Multiway_Trees A.18.10(31/5)
in Ada.Containers.Ordered_Maps A.18.6(16.4/5), A.18.6(16.6/5)
in Ada.Containers.Vectors A.18.2(34.4/5), A.18.2(34.6/5)
in Ada.Interrupts C.3.2(10)
in Ada.Task_Attributes C.7.2(5)
Reference_Preserving_Key
in Ada.Containers.Hashed_Sets A.18.8(58.2/5), A.18.8(58.4/5)
in Ada.Containers.Ordered_Sets A.18.9(73.2/5), A.18.9(73.4/5)
Reinitialize *in* Ada.Task_Attributes C.7.2(6)
Relative_Name
in Ada.Directories.Hierarchical_File_Names A.16.1(13/3)
Remove_Task
in Ada.Execution_Time.Group_Budgets D.14.2(8/2)
Rename *in* Ada.Directories A.16(12/2)
Replace
in Ada.Containers.Hashed_Maps A.18.5(23/5)
in Ada.Containers.Hashed_Sets A.18.8(22/5), A.18.8(53/5)
in Ada.Containers.Ordered_Maps A.18.6(22/5)
in Ada.Containers.Ordered_Sets A.18.9(21/5), A.18.9(66/5)
Replace_Element
in Ada.Containers.Doubly_Linked_Lists A.18.3(15/5)
in Ada.Containers.Hashed_Maps A.18.5(15/5)
in Ada.Containers.Hashed_Sets A.18.8(16/5)
in Ada.Containers.Indefinite_Holders A.18.18(13/5)
in Ada.Containers.Multiway_Trees A.18.10(25/5)
in Ada.Containers.Ordered_Maps A.18.6(14/5)
in Ada.Containers.Ordered_Sets A.18.9(15/5)
in Ada.Containers.Vectors A.18.2(29/5), A.18.2(30/5)
in Ada.Strings.Bounded A.4.4(27)
in Ada.Strings.Unbounded A.4.5(21)
Replace_Slice
in Ada.Strings.Bounded A.4.4(58), A.4.4(59)
in Ada.Strings.Fixed A.4.3(23), A.4.3(24)

in Ada.Strings.Unbounded A.4.5(53), A.4.5(54)
Replenish
in Ada.Execution_Time.Group_Budgets D.14.2(9/2)
Replicate *in* Ada.Strings.Bounded A.4.4(78), A.4.4(79), A.4.4(80)
Reraise_Occurrence *in* Ada.Exceptions 11.4.1(4/3)
Reserve_Capacity
in Ada.Containers.Hashed_Maps A.18.5(9/5)
in Ada.Containers.Hashed_Sets A.18.8(11/5)
in Ada.Containers.Vectors A.18.2(20/5)
Reset
in Ada.Direct_IO A.8.4(8)
in Ada.Numerics.Discrete_Random A.5.2(21/5), A.5.2(24/5)
in Ada.Numerics.Float_Random A.5.2(9/5), A.5.2(12/5)
in Ada.Sequential_IO A.8.1(8)
in Ada.Streams.Stream_IO A.12.1(10)
in Ada.Text_IO A.10.1(11)
Reverse_Elements
in Ada.Containers.Doubly_Linked_Lists A.18.3(27/5)
in Ada.Containers.Vectors A.18.2(54/5)
Reverse_Find
in Ada.Containers.Doubly_Linked_Lists A.18.3(42/5)
in Ada.Containers.Vectors A.18.2(70/5)
Reverse_Find_Index
in Ada.Containers.Vectors A.18.2(69/2)
Reverse_Iterate
in Ada.Containers.Doubly_Linked_Lists A.18.3(46/5)
in Ada.Containers.Ordered_Maps A.18.6(51/5)
in Ada.Containers.Ordered_Sets A.18.9(61/5)
in Ada.Containers.Vectors A.18.2(74/5)
Reverse_Iterate_Children
in Ada.Containers.Multiway_Trees A.18.10(69.1/5), A.18.10(69/5)
Root *in* Ada.Containers.Multiway_Trees A.18.10(22/5)
Save
in Ada.Numerics.Discrete_Random A.5.2(24/5)
in Ada.Numerics.Float_Random A.5.2(12/5)
Save_Occurrence *in* Ada.Exceptions 11.4.1(6/2)
Second *in* Ada.Calendar.Formatting 9.6.1(26/2)
Seconds
in Ada.Calendar 9.6(13)
in Ada.Real_Time D.8(14/2)
Seconds_Of *in* Ada.Calendar.Formatting 9.6.1(28/2)
Set *in* Ada.Environment_Variables A.17(6/2)
Set_Bounded_String
in Ada.Strings.Bounded A.4.4(12.1/2)
Set_Col *in* Ada.Text_IO A.10.1(35/5)
Set_CPU
in System.Multiprocessors.Dispatching_Domains D.16.1(12/3)
Set_Deadline *in* Ada.Dispatching.EDF D.2.6(9/5)
Set_Dependents_Fallback_Handler
in Ada.Task_Termination C.7.3(5/2)
Set_Directory *in* Ada.Directories A.16(6/2)
Set_Error *in* Ada.Text_IO A.10.1(15)
Set_Exit_Status *in* Ada.Command_Line A.15(9)
Set_False
in Ada.Synchronous_Task_Control D.10(4/5)
Set_Handler
in Ada.Execution_Time.Group_Budgets D.14.2(10/2)
in Ada.Execution_Time.Timers D.14.1(7/2)
in Ada.Real_Time.Timing_Events D.15(5/2)
Set_Im
in Ada.Numerics.Generic_Complex_Arrays G.3.2(8/2), G.3.2(28/2)
in Ada.Numerics.Generic_Complex_Types G.1.1(7)
Set_Index
in Ada.Direct_IO A.8.4(14/5)
in Ada.Streams.Stream_IO A.12.1(22/5)
Set_Input *in* Ada.Text_IO A.10.1(15)
Set_Length *in* Ada.Containers.Vectors A.18.2(22/5)
Set_Line *in* Ada.Text_IO A.10.1(36/5)
Set_Line_Length *in* Ada.Text_IO A.10.1(23/5)
Set_Mode *in* Ada.Streams.Stream_IO A.12.1(24)
Set_Output *in* Ada.Text_IO A.10.1(15)
Set_Page_Length *in* Ada.Text_IO A.10.1(24/5)
Set_Pool_of_Subpool
in System.Storage_Pools.Subpools 13.11.4(10/5)
Set_Priority
in Ada.Dynamic_Priorities D.5.1(4)
Set_Quantum
in Ada.Dispatching.Round_Robin D.2.5(4/5)
Set_Re
in Ada.Numerics.Generic_Complex_Arrays G.3.2(8/2), G.3.2(28/2)
in Ada.Numerics.Generic_Complex_Types G.1.1(7)
Set_Relative_Deadline
in Ada.Dispatching.EDF D.2.6(9/5)
Set_Specific_Handler
in Ada.Task_Termination C.7.3(6/2)
Set_True
in Ada.Synchronous_Task_Control D.10(4/5)
Set_Unbounded_String
in Ada.Strings.Unbounded A.4.5(11.1/2)
Set_Value *in* Ada.Task_Attributes C.7.2(6)
Simple_Name
in Ada.Directories A.16(16/5), A.16(38/2)
in Ada.Directories.Hierarchical_File_Names A.16.1(10/3)
Sin
in Ada.Numerics.Generic_Complex_Elementary_Functions G.1.2(4)
in Ada.Numerics.Generic_Elementary_Functions A.5.1(5)
Sinh
in Ada.Numerics.Generic_Complex_Elementary_Functions G.1.2(6)
in Ada.Numerics.Generic_Elementary_Functions A.5.1(7)
Size
in Ada.Direct_IO A.8.4(15)
in Ada.Directories A.16(26/2), A.16(41/2)
in Ada.Streams.Stream_IO A.12.1(23)
Skip_Line *in* Ada.Text_IO A.10.1(29/5)
Skip_Page *in* Ada.Text_IO A.10.1(32/5)
Slice
in Ada.Strings.Bounded A.4.4(28)
in Ada.Strings.Unbounded A.4.5(22)
Solve
in Ada.Numerics.Generic_Complex_Arrays G.3.2(46/2)
in Ada.Numerics.Generic_Real_Arrays G.3.1(24/2)
Sort
in Ada.Containers.Doubly_Linked_Lists A.18.3(49/5)
in Ada.Containers.Vectors A.18.2(77/5)
Specific_Handler
in Ada.Task_Termination C.7.3(6/2)
Splice
in Ada.Containers.Doubly_Linked_Lists A.18.3(30/5), A.18.3(31/5), A.18.3(32/2)

Splice_Children
in Ada.Containers.Multiway_Trees A.18.10(57/5), A.18.10(58/5)
Splice_Subtree
in Ada.Containers.Multiway_Trees A.18.10(55/5), A.18.10(56/5)
Split
in Ada.Calendar 9.6(14)
in Ada.Calendar.Formatting 9.6.1(29/2), 9.6.1(32/2), 9.6.1(33/2), 9.6.1(34/2)
in Ada.Execution_Time D.14(8/2)
in Ada.Real_Time D.8(16)
Split_Into_Chunks
in Ada.Iterator_Interfaces 5.5.1(4.4/5)
Sqrt
in Ada.Numerics.Generic_Complex_Elementary_Functions G.1.2(3)
in Ada.Numerics.Generic_Elementary_Functions A.5.1(4)
Standard_Error *in* Ada.Text_IO A.10.1(16), A.10.1(19)
Standard_Input *in* Ada.Text_IO A.10.1(16), A.10.1(19)
Standard_Output *in* Ada.Text_IO A.10.1(16), A.10.1(19)
Start_Search *in* Ada.Directories A.16(32/2)
Storage_Size
in System.Storage_Pools 13.11(9)
in System.Storage_Pools.Subpools 13.11.4(16/3)
Stream
in Ada.Streams.Stream_IO A.12.1(13)
in Ada.Text_IO.Text_Streams A.12.2(4)
in Ada.Wide_Text_IO.Text_Streams A.12.3(4)
in Ada.Wide_Wide_Text_IO.Text_Streams A.12.4(4/2)
Strlen *in* Interfaces.C.Strings B.3.1(17)
Sub_Second *in* Ada.Calendar.Formatting 9.6.1(27/2)
Subtree_Node_Count
in Ada.Containers.Multiway_Trees A.18.10(18.1/5), A.18.10(18/5)
Supported
in Ada.Execution_Time.Interrupts D.14.3(3/5)
Suspend_Until_True
in Ada.Synchronous_Task_Control D.10(4/5)
Suspend_Until_True_And_Set_Deadline
in Ada.Synchronous_Task_Control.EDF D.10(5.2/5)
Swap
in Ada.Containers.Doubly_Linked_Lists A.18.3(28/5)
in Ada.Containers.Indefinite_Holders A.18.18(22.1/5)
in Ada.Containers.Multiway_Trees A.18.10(37/5)
in Ada.Containers.Vectors A.18.2(55/5), A.18.2(56/5)
Swap_Links
in Ada.Containers.Doubly_Linked_Lists A.18.3(29/5)
Symmetric_Difference
in Ada.Containers.Hashed_Sets A.18.8(35/5), A.18.8(36/5)
in Ada.Containers.Ordered_Sets A.18.9(36/5), A.18.9(37/5)
Tail
in Ada.Strings.Bounded A.4.4(72), A.4.4(73)
in Ada.Strings.Fixed A.4.3(37), A.4.3(38)
in Ada.Strings.Unbounded A.4.5(67), A.4.5(68)
Tampering_With_Cursors_Prohibited
in Ada.Containers.Doubly_Linked_Lists A.18.3(10.1/5)
in Ada.Containers.Hashed_Maps A.18.5(7.1/5)
in Ada.Containers.Hashed_Sets A.18.8(8.1/5)
in Ada.Containers.Multiway_Trees A.18.10(15.1/5)
in Ada.Containers.Ordered_Maps A.18.6(8.1/5)
in Ada.Containers.Ordered_Sets A.18.9(9.1/5)
in Ada.Containers.Vectors A.18.2(12.1/5)
Tampering_With_Elements_Prohibited
in Ada.Containers.Doubly_Linked_Lists A.18.3(10.2/5)
in Ada.Containers.Hashed_Maps A.18.5(7.2/5)
in Ada.Containers.Multiway_Trees A.18.10(15.2/5)
in Ada.Containers.Ordered_Maps A.18.6(8.2/5)
in Ada.Containers.Vectors A.18.2(12.2/5)
Tampering_With_The_Element_Prohibited
in Ada.Containers.Indefinite_Holders A.18.18(8.1/5)
Tan
in Ada.Numerics.Generic_Complex_Elementary_Functions G.1.2(4)
in Ada.Numerics.Generic_Elementary_Functions A.5.1(5)
Tanh
in Ada.Numerics.Generic_Complex_Elementary_Functions G.1.2(6)
in Ada.Numerics.Generic_Elementary_Functions A.5.1(7)
Text_Truncated
in Ada.Strings.Text_Buffers.Bounded A.4.12(28/5)
Time_Of
in Ada.Calendar 9.6(15)
in Ada.Calendar.Formatting 9.6.1(30/2), 9.6.1(31/2)
in Ada.Execution_Time D.14(9/2)
in Ada.Real_Time D.8(16)
Time_Of_Event
in Ada.Real_Time.Timing_Events D.15(6/2)
Time_Remaining
in Ada.Execution_Time.Timers D.14.1(8/2)
To_Ada
in Interfaces.C B.3(22), B.3(26), B.3(28), B.3(32), B.3(37), B.3(39), B.3(39.10/2), B.3(39.13/2), B.3(39.17/2), B.3(39.19/2), B.3(39.4/2), B.3(39.8/2)
in Interfaces.COBOL B.4(17), B.4(19)
in Interfaces.Fortran B.5(13), B.5(14), B.5(16)
To_Address
in System.Address_To_Access_Conversions 13.7.2(3/3)
in System.Storage_Elements 13.7.1(10/3)
To_Basic
in Ada.Characters.Handling A.3.2(6), A.3.2(7)
in Ada.Wide_Characters.Handling A.3.5(20.1/5), A.3.5(21.1/5)
To_Big_Integer
in Ada.Numerics.Big_Numbers.Big_Integers A.5.6(7/5), A.5.6(12/5), A.5.6(13/5)
To_Big_Real
in Ada.Numerics.Big_Numbers.Big_Reals A.5.7(9/5), A.5.7(13/5), A.5.7(14/5)
To_Binary *in* Interfaces.COBOL B.4(45), B.4(48)
To_Bounded_String
in Ada.Strings.Bounded A.4.4(11)
To_C *in* Interfaces.C B.3(21), B.3(25), B.3(27), B.3(32), B.3(36), B.3(38), B.3(39.13/2), B.3(39.16/2), B.3(39.18/2), B.3(39.4/2), B.3(39.7/2), B.3(39.9/2)
To_Character
in Ada.Characters.Conversions A.3.4(5/2)
To_Chars_Ptr *in* Interfaces.C.Strings B.3.1(8)
To_COBOL *in* Interfaces.COBOL B.4(17), B.4(18)
To_Cursor *in* Ada.Containers.Vectors A.18.2(25/5)
To_Decimal *in* Interfaces.COBOL B.4(35), B.4(40), B.4(44), B.4(47)
To_Display *in* Interfaces.COBOL B.4(36)
To_Domain
in Ada.Strings.Maps A.4.2(24)
in Ada.Strings.Wide_Maps A.4.7(24)
in Ada.Strings.Wide_Wide_Maps A.4.8(24/2)

To_Duration *in* Ada.Real_Time D.8(13)
To_Fortran *in* Interfaces.Fortran B.5(13), B.5(14), B.5(15)
To_Holder
  in Ada.Containers.Indefinite_Holders A.18.18(9/5)
To_Index *in* Ada.Containers.Vectors A.18.2(26.1/5), A.18.2(26/5)
To_Integer
  in Ada.Numerics.Big_Numbers.Big_Integers A.5.6(11/5)
  in System.Storage_Elements 13.7.1(10/3)
To_ISO_646 *in* Ada.Characters.Handling A.3.2(11), A.3.2(12)
To_Long_Binary *in* Interfaces.COBOL B.4(48)
To_Lower
  in Ada.Characters.Handling A.3.2(6), A.3.2(7)
  in Ada.Wide_Characters.Handling A.3.5(20/3), A.3.5(21/3)
To_Mapping
  in Ada.Strings.Maps A.4.2(23)
  in Ada.Strings.Wide_Maps A.4.7(23)
  in Ada.Strings.Wide_Wide_Maps A.4.8(23/2)
To_Packed *in* Interfaces.COBOL B.4(41)
To_Picture *in* Ada.Text_IO.Editing F.3.3(6)
To_Pointer
  in System.Address_To_Access_Conversions 13.7.2(3/3)
To_Quotient_String
  in Ada.Numerics.Big_Numbers.Big_Reals A.5.7(19/5)
To_Range
  in Ada.Strings.Maps A.4.2(24)
  in Ada.Strings.Wide_Maps A.4.7(25)
  in Ada.Strings.Wide_Wide_Maps A.4.8(25/2)
To_Ranges
  in Ada.Strings.Maps A.4.2(10)
  in Ada.Strings.Wide_Maps A.4.7(10)
  in Ada.Strings.Wide_Wide_Maps A.4.8(10/2)
To_Real
  in Ada.Numerics.Big_Numbers.Big_Reals A.5.7(10/5)
To_Sequence
  in Ada.Strings.Maps A.4.2(19)
  in Ada.Strings.Wide_Maps A.4.7(19)
  in Ada.Strings.Wide_Wide_Maps A.4.8(19/2)
To_Set
  in Ada.Containers.Hashed_Sets A.18.8(9/5)
  in Ada.Containers.Ordered_Sets A.18.9(10/5)
  in Ada.Strings.Maps A.4.2(8), A.4.2(9), A.4.2(17), A.4.2(18)
  in Ada.Strings.Wide_Maps A.4.7(8), A.4.7(9), A.4.7(17), A.4.7(18)
  in Ada.Strings.Wide_Wide_Maps A.4.8(8/2), A.4.8(9/2), A.4.8(17/2), A.4.8(18/2)
To_String
  in Ada.Characters.Conversions A.3.4(5/2)
  in Ada.Numerics.Big_Numbers.Big_Integers A.5.6(14/5)
  in Ada.Numerics.Big_Numbers.Big_Reals A.5.7(15/5)
  in Ada.Strings.Bounded A.4.4(12)
  in Ada.Strings.Unbounded A.4.5(11)
To_Time_Span *in* Ada.Real_Time D.8(13)
To_Unbounded_String
  in Ada.Strings.Unbounded A.4.5(9), A.4.5(10)
To_Upper
  in Ada.Characters.Handling A.3.2(6), A.3.2(7)
  in Ada.Wide_Characters.Handling A.3.5(20/3), A.3.5(21/3)
To_Vector *in* Ada.Containers.Vectors A.18.2(13/5), A.18.2(14/5)
To_Wide_Character
  in Ada.Characters.Conversions A.3.4(4/2), A.3.4(5/2)
To_Wide_String
  in Ada.Characters.Conversions A.3.4(4/2), A.3.4(5/2)
To_Wide_Wide_Character
  in Ada.Characters.Conversions A.3.4(4/2)
To_Wide_Wide_String
  in Ada.Characters.Conversions A.3.4(4/2)
Translate
  in Ada.Strings.Bounded A.4.4(53), A.4.4(54), A.4.4(55), A.4.4(56)
  in Ada.Strings.Fixed A.4.3(18), A.4.3(19), A.4.3(20), A.4.3(21)
  in Ada.Strings.Unbounded A.4.5(48), A.4.5(49), A.4.5(50), A.4.5(51)
Transpose
  in Ada.Numerics.Generic_Complex_Arrays G.3.2(34/2)
  in Ada.Numerics.Generic_Real_Arrays G.3.1(17/2)
Trim
  in Ada.Strings.Bounded A.4.4(67), A.4.4(68), A.4.4(69)
  in Ada.Strings.Fixed A.4.3(31), A.4.3(32), A.4.3(33), A.4.3(34)
  in Ada.Strings.Unbounded A.4.5(61), A.4.5(62), A.4.5(63), A.4.5(64)
Unbounded_Slice
  in Ada.Strings.Unbounded A.4.5(22.1/2), A.4.5(22.2/2)
Unchecked_Conversion
  child of Ada 13.9(3/5)
Unchecked_Deallocation
  child of Ada 13.11.2(3/5)
Union
  in Ada.Containers.Hashed_Sets A.18.8(26/5), A.18.8(27/5)
  in Ada.Containers.Ordered_Sets A.18.9(27/5), A.18.9(28/5)
Unit_Matrix
  in Ada.Numerics.Generic_Complex_Arrays G.3.2(51/2)
  in Ada.Numerics.Generic_Real_Arrays G.3.1(29/2)
Unit_Vector
  in Ada.Numerics.Generic_Complex_Arrays G.3.2(24/2)
  in Ada.Numerics.Generic_Real_Arrays G.3.1(14/2)
Update *in* Interfaces.C.Strings B.3.1(18), B.3.1(19)
Update_Element
  in Ada.Containers.Doubly_Linked_Lists A.18.3(17/5)
  in Ada.Containers.Hashed_Maps A.18.5(17/5)
  in Ada.Containers.Indefinite_Holders A.18.18(15/5)
  in Ada.Containers.Multiway_Trees A.18.10(27/5)
  in Ada.Containers.Ordered_Maps A.18.6(16/5)
  in Ada.Containers.Vectors A.18.2(33/5), A.18.2(34/5)
Update_Element_Preserving_Key
  in Ada.Containers.Hashed_Sets A.18.8(58/5)
  in Ada.Containers.Ordered_Sets A.18.9(73/5)
Update_Error *in* Interfaces.C.Strings B.3.1(20)
UTC_Time_Offset
  in Ada.Calendar.Time_Zones 9.6.1(6.1/5)
Valid
  in Ada.Text_IO.Editing F.3.3(5), F.3.3(12)
  in Interfaces.COBOL B.4(33), B.4(38), B.4(43)
Value
  in Ada.Calendar.Formatting 9.6.1(36/2), 9.6.1(38/2)
  in Ada.Environment_Variables A.17(4.1/3), A.17(4/2)
  in Ada.Numerics.Discrete_Random A.5.2(26)
  in Ada.Numerics.Float_Random A.5.2(14)
  in Ada.Strings.Maps A.4.2(21)
  in Ada.Strings.Wide_Maps A.4.7(21)
  in Ada.Strings.Wide_Wide_Maps A.4.8(21/2)
  in Ada.Task_Attributes C.7.2(4)
  in Interfaces.C.Pointers B.3.2(6), B.3.2(7)
  in Interfaces.C.Strings B.3.1(13), B.3.1(14), B.3.1(15), B.3.1(16)

Virtual_Length
in Interfaces.C.Pointers B.3.2(13)
Wait_For_Release
in Ada.Synchronous_Barriers D.10.1(6/5)
Wide_Equal_Case_Insensitive
child of Ada.Strings.Wide_Bounded A.4.7(1/3)
child of Ada.Strings.Wide_Fixed A.4.7(1/3)
child of Ada.Strings.Wide_Unbounded A.4.7(1/3)
Wide_Hash
child of Ada.Strings.Wide_Bounded A.4.7(1/3)
child of Ada.Strings.Wide_Fixed A.4.7(1/3)
child of Ada.Strings.Wide_Unbounded A.4.7(1/3)
Wide_Hash_Case_Insensitive
child of Ada.Strings.Wide_Bounded A.4.7(1/3)
child of Ada.Strings.Wide_Fixed A.4.7(1/3)
child of Ada.Strings.Wide_Unbounded A.4.7(1/3)
Wide_Exception_Name *in* Ada.Exceptions 11.4.1(2/5), 11.4.1(5/2)
Wide_Expanded_Name *in* Ada.Tags 3.9(7/2)
Wide_Get
in Ada.Strings.Text_Buffers.Unbounded A.4.12(21/5)
Wide_Get_UTF_16
in Ada.Strings.Text_Buffers.Unbounded A.4.12(24/5)
Wide_Put *in* Ada.Strings.Text_Buffers A.4.12(8/5)
Wide_Put_UTF_16
in Ada.Strings.Text_Buffers A.4.12(11/5)
Wide_Wide_Equal_Case_Insensitive
child of Ada.Strings.Wide_Wide_Bounded A.4.8(1/3)
child of Ada.Strings.Wide_Wide_Fixed A.4.8(1/3)
child of Ada.Strings.Wide_Wide_Unbounded A.4.8(1/3)
Wide_Wide_Hash
child of Ada.Strings.Wide_Wide_Bounded A.4.8(1/3)
child of Ada.Strings.Wide_Wide_Fixed A.4.8(1/3)
child of Ada.Strings.Wide_Wide_Unbounded A.4.8(1/3)
Wide_Wide_Hash_Case_Insensitive
child of Ada.Strings.Wide_Wide_Bounded A.4.8(1/3)
child of Ada.Strings.Wide_Wide_Fixed A.4.8(1/3)
child of Ada.Strings.Wide_Wide_Unbounded A.4.8(1/3)
Wide_Wide_Exception_Name
in Ada.Exceptions 11.4.1(2/5), 11.4.1(5/2)
Wide_Wide_Expanded_Name *in* Ada.Tags 3.9(7/2)
Wide_Wide_Get
in Ada.Strings.Text_Buffers.Unbounded A.4.12(22/5)
Wide_Wide_Put
in Ada.Strings.Text_Buffers A.4.12(9/5)
Write
in Ada.Direct_IO A.8.4(13/5)
in Ada.Sequential_IO A.8.1(12/5)
in Ada.Storage_IO A.9(7)
in Ada.Streams 13.13.1(6)
in Ada.Streams.Storage.Bounded 13.13.1(29/5)
in Ada.Streams.Storage.Unbounded 13.13.1(21/5)
in Ada.Streams.Stream_IO A.12.1(18/5), A.12.1(19/5)
in System.RPC E.5(8)
Year
in Ada.Calendar 9.6(13)
in Ada.Calendar.Formatting 9.6.1(21/2)
Yield *in* Ada.Dispatching D.2.1(1.3/5)
Yield_To_Higher
in Ada.Dispatching.Non_Preemptive D.2.4(2.2/5)
Yield_To_Same_Or_Higher
in Ada.Dispatching.Non_Preemptive D.2.4(2.2/5)

Q.4 Language-Defined Exceptions

{*AI95-00440-01*} {*AI05-0299-1*} This subclause lists all language-defined exceptions. 1/3

Argument_Error
in Ada.Numerics A.5(3/5)
Assertion_Error
in Ada.Assertions 11.4.2(13/2)
Capacity_Error
in Ada.Containers A.18.1(5.1/3)
Communication_Error
in System.RPC E.5(5)
Constraint_Error
in Standard A.1(46)
Conversion_Error
in Interfaces.COBOL B.4(30)
Data_Error
in Ada.Direct_IO A.8.4(18)
in Ada.IO_Exceptions A.13(4)
in Ada.Sequential_IO A.8.1(15)
in Ada.Storage_IO A.9(9)
in Ada.Streams.Stream_IO A.12.1(26)
in Ada.Text_IO A.10.1(85/5)
Device_Error
in Ada.Direct_IO A.8.4(18)
in Ada.Directories A.16(43/2)
in Ada.IO_Exceptions A.13(4)
in Ada.Sequential_IO A.8.1(15)
in Ada.Streams.Stream_IO A.12.1(26)
in Ada.Text_IO A.10.1(85/5)
Dispatching_Domain_Error
in System.Multiprocessors.Dispatching_Domains D.16.1(4/3)
Dispatching_Policy_Error
in Ada.Dispatching D.2.1(1.4/3)
Encoding_Error
in Ada.Strings.UTF_Encoding A.4.11(8/3)
End_Error
in Ada.Direct_IO A.8.4(18)
in Ada.IO_Exceptions A.13(4)
in Ada.Sequential_IO A.8.1(15)
in Ada.Streams.Stream_IO A.12.1(26)
in Ada.Text_IO A.10.1(85/5)
Group_Budget_Error
in Ada.Execution_Time.Group_Budgets D.14.2(11/2)
Index_Error
in Ada.Strings A.4.1(5)
Layout_Error
in Ada.IO_Exceptions A.13(4)
in Ada.Text_IO A.10.1(85/5)
Length_Error
in Ada.Strings A.4.1(5)

Mode_Error
in Ada.Direct_IO A.8.4(18)
in Ada.IO_Exceptions A.13(4)
in Ada.Sequential_IO A.8.1(15)
in Ada.Streams.Stream_IO A.12.1(26)
in Ada.Text_IO A.10.1(85/5)
Name_Error
in Ada.Direct_IO A.8.4(18)
in Ada.Directories A.16(43/2)
in Ada.IO_Exceptions A.13(4)
in Ada.Sequential_IO A.8.1(15)
in Ada.Streams.Stream_IO A.12.1(26)
in Ada.Text_IO A.10.1(85/5)
Pattern_Error
in Ada.Strings A.4.1(5)
Picture_Error
in Ada.Text_IO.Editing F.3.3(9)
Pointer_Error
in Interfaces.C.Pointers B.3.2(8)
Program_Error
in Standard A.1(46)
Status_Error
in Ada.Direct_IO A.8.4(18)
in Ada.Directories A.16(43/2)
in Ada.IO_Exceptions A.13(4)
in Ada.Sequential_IO A.8.1(15)
in Ada.Streams.Stream_IO A.12.1(26)
in Ada.Text_IO A.10.1(85/5)
Storage_Error
in Standard A.1(46)
Tag_Error
in Ada.Tags 3.9(8)
Tasking_Error
in Standard A.1(46)
Terminator_Error
in Interfaces.C B.3(40)
Time_Error
in Ada.Calendar 9.6(18)
Timer_Resource_Error
in Ada.Execution_Time.Timers D.14.1(9/2)
Translation_Error
in Ada.Strings A.4.1(5)
Unknown_Zone_Error
in Ada.Calendar.Time_Zones 9.6.1(5/2)
Use_Error
in Ada.Direct_IO A.8.4(18)
in Ada.Directories A.16(43/2)
in Ada.IO_Exceptions A.13(4)
in Ada.Sequential_IO A.8.1(15)
in Ada.Streams.Stream_IO A.12.1(26)
in Ada.Text_IO A.10.1(85/5)

Q.5 Language-Defined Objects

1/3 {*AI95-00440-01*} {*AI05-0299-1*} This subclause lists all language-defined constants, variables, named numbers, and enumeration literals.

1.a/2 **To be honest:** Formally, named numbers and enumeration literals aren't objects, but it was thought to be too weird to say “Language-Defined Objects and Values”.

ACK *in* Ada.Characters.Latin_1 A.3.3(5)
Acute *in* Ada.Characters.Latin_1 A.3.3(22)
Ada_To_COBOL *in* Interfaces.COBOL B.4(14)
Alphanumeric_Set
in Ada.Strings.Maps.Constants A.4.6(4)
Ampersand *in* Ada.Characters.Latin_1 A.3.3(8)
APC *in* Ada.Characters.Latin_1 A.3.3(19)
Apostrophe *in* Ada.Characters.Latin_1 A.3.3(8)
Asterisk *in* Ada.Characters.Latin_1 A.3.3(8)
Basic_Map
in Ada.Strings.Maps.Constants A.4.6(5)
Basic_Set
in Ada.Strings.Maps.Constants A.4.6(4)
BEL *in* Ada.Characters.Latin_1 A.3.3(5)
BOM_16 *in* Ada.Strings.UTF_Encoding A.4.11(12/3)
BOM_16BE *in* Ada.Strings.UTF_Encoding A.4.11(10/3)
BOM_16LE *in* Ada.Strings.UTF_Encoding A.4.11(11/3)
BOM_8 *in* Ada.Strings.UTF_Encoding A.4.11(9/3)
BPH *in* Ada.Characters.Latin_1 A.3.3(17)
Broken_Bar *in* Ada.Characters.Latin_1 A.3.3(21/3)
BS *in* Ada.Characters.Latin_1 A.3.3(5)
Buffer_Size *in* Ada.Storage_IO A.9(4)
CAN *in* Ada.Characters.Latin_1 A.3.3(6)
CCH *in* Ada.Characters.Latin_1 A.3.3(18)
Cedilla *in* Ada.Characters.Latin_1 A.3.3(22)
Cent_Sign *in* Ada.Characters.Latin_1 A.3.3(21/3)
char16_nul *in* Interfaces.C B.3(39.3/2)
char32_nul *in* Interfaces.C B.3(39.12/2)
CHAR_BIT *in* Interfaces.C B.3(6)
Character_Set
in Ada.Strings.Wide_Maps A.4.7(46/2)
in Ada.Strings.Wide_Maps.Wide_Constants A.4.8(48/2)
Circumflex *in* Ada.Characters.Latin_1 A.3.3(12)
COBOL_To_Ada *in* Interfaces.COBOL B.4(15)
Colon *in* Ada.Characters.Latin_1 A.3.3(10)
Comma *in* Ada.Characters.Latin_1 A.3.3(8)
Commercial_At
in Ada.Characters.Latin_1 A.3.3(10)
Control_Set
in Ada.Strings.Maps.Constants A.4.6(4)
Copyright_Sign
in Ada.Characters.Latin_1 A.3.3(21/3)
Country_Unknown *in* Ada.Locales A.19(5/3)
CPU_Tick *in* Ada.Execution_Time D.14(4/2)
CPU_Time_First *in* Ada.Execution_Time D.14(4/2)
CPU_Time_Last *in* Ada.Execution_Time D.14(4/2)
CPU_Time_Unit *in* Ada.Execution_Time D.14(4/2)
CR *in* Ada.Characters.Latin_1 A.3.3(5)
CSI *in* Ada.Characters.Latin_1 A.3.3(19)
Currency_Sign
in Ada.Characters.Latin_1 A.3.3(21/3)
DC1 *in* Ada.Characters.Latin_1 A.3.3(6)

DC2 *in* Ada.Characters.Latin_1 A.3.3(6)
DC3 *in* Ada.Characters.Latin_1 A.3.3(6)
DC4 *in* Ada.Characters.Latin_1 A.3.3(6)
DCS *in* Ada.Characters.Latin_1 A.3.3(18)
Decimal_Digit_Set
in Ada.Strings.Maps.Constants A.4.6(4)
Default_Aft
in Ada.Text_IO A.10.1(64), A.10.1(69), A.10.1(74)
in Ada.Text_IO.Complex_IO G.1.3(5)
Default_Base *in* Ada.Text_IO A.10.1(53), A.10.1(58)
Default_Bit_Order *in* System 13.7(15/2)
Default_Currency
in Ada.Text_IO.Editing F.3.3(10)
Default_Deadline
in Ada.Dispatching.EDF D.2.6(9/5)
Default_Exp
in Ada.Text_IO A.10.1(64), A.10.1(69), A.10.1(74)
in Ada.Text_IO.Complex_IO G.1.3(5)
Default_Fill *in* Ada.Text_IO.Editing F.3.3(10)
Default_Fore
in Ada.Text_IO A.10.1(64), A.10.1(69), A.10.1(74)
in Ada.Text_IO.Complex_IO G.1.3(5)
Default_Priority *in* System 13.7(17)
Default_Quantum
in Ada.Dispatching.Round_Robin D.2.5(4/5)
Default_Radix_Mark
in Ada.Text_IO.Editing F.3.3(10)
Default_Relative_Deadline
in Ada.Dispatching.EDF D.2.6(9/5)
Default_Separator
in Ada.Text_IO.Editing F.3.3(10)
Default_Setting *in* Ada.Text_IO A.10.1(80)
Default_Width *in* Ada.Text_IO A.10.1(53), A.10.1(58), A.10.1(80)
Degree_Sign *in* Ada.Characters.Latin_1 A.3.3(22)
DEL *in* Ada.Characters.Latin_1 A.3.3(14)
Diaeresis *in* Ada.Characters.Latin_1 A.3.3(21/3)
Division_Sign
in Ada.Characters.Latin_1 A.3.3(26)
DLE *in* Ada.Characters.Latin_1 A.3.3(6)
Dollar_Sign *in* Ada.Characters.Latin_1 A.3.3(8)
e *in* Ada.Numerics A.5(3/5)
EM *in* Ada.Characters.Latin_1 A.3.3(6)
Empty_Holder
in Ada.Containers.Indefinite_Holders A.18.18(7/3)
Empty_List
in Ada.Containers.Doubly_Linked_Lists A.18.3(8/2), A.18.3(51.4/5)
Empty_Map
in Ada.Containers.Hashed_Maps A.18.5(5/2), A.18.5(37.5/5)
in Ada.Containers.Ordered_Maps A.18.6(6/2), A.18.6(51.6/5)
Empty_Set
in Ada.Containers.Hashed_Sets A.18.8(5/2), A.18.8(59.4/5)
in Ada.Containers.Ordered_Sets A.18.9(6/2), A.18.9(74.4/5)
Empty_Tree
in Ada.Containers.Multiway_Trees A.18.10(10/3), A.18.10(70.4/5)
Empty_Vector
in Ada.Containers.Vectors A.18.2(10/2), A.18.2(79.4/5)
ENQ *in* Ada.Characters.Latin_1 A.3.3(5)
EOT *in* Ada.Characters.Latin_1 A.3.3(5)
EPA *in* Ada.Characters.Latin_1 A.3.3(18)
Equals_Sign *in* Ada.Characters.Latin_1 A.3.3(10)
ESA *in* Ada.Characters.Latin_1 A.3.3(17)
ESC *in* Ada.Characters.Latin_1 A.3.3(6)
ETB *in* Ada.Characters.Latin_1 A.3.3(6)
ETX *in* Ada.Characters.Latin_1 A.3.3(5)
Exclamation *in* Ada.Characters.Latin_1 A.3.3(8)
Failure *in* Ada.Command_Line A.15(8)
Feminine_Ordinal_Indicator
in Ada.Characters.Latin_1 A.3.3(21/3)
FF *in* Ada.Characters.Latin_1 A.3.3(5)
Fine_Delta *in* System 13.7(9)
Fraction_One_Half
in Ada.Characters.Latin_1 A.3.3(22)
Fraction_One_Quarter
in Ada.Characters.Latin_1 A.3.3(22)
Fraction_Three_Quarters
in Ada.Characters.Latin_1 A.3.3(22)
Friday *in* Ada.Calendar.Formatting 9.6.1(17/2)
FS *in* Ada.Characters.Latin_1 A.3.3(6)
Full_Stop *in* Ada.Characters.Latin_1 A.3.3(8)
Graphic_Set
in Ada.Strings.Maps.Constants A.4.6(4)
Grave *in* Ada.Characters.Latin_1 A.3.3(13)
Greater_Than_Sign
in Ada.Characters.Latin_1 A.3.3(10)
GS *in* Ada.Characters.Latin_1 A.3.3(6)
Hexadecimal_Digit_Set
in Ada.Strings.Maps.Constants A.4.6(4)
High_Order_First
in Interfaces.COBOL B.4(25)
in System 13.7(15/2)
HT *in* Ada.Characters.Latin_1 A.3.3(5)
HTJ *in* Ada.Characters.Latin_1 A.3.3(17)
HTS *in* Ada.Characters.Latin_1 A.3.3(17)
Hyphen *in* Ada.Characters.Latin_1 A.3.3(8)
i
in Ada.Numerics.Generic_Complex_Types G.1.1(5)
in Interfaces.Fortran B.5(10)
Identity
in Ada.Strings.Maps A.4.2(22)
in Ada.Strings.Wide_Maps A.4.7(22)
in Ada.Strings.Wide_Wide_Maps A.4.8(22/2)
Interrupt_Clocks_Supported
in Ada.Execution_Time D.14(9.1/3)
Inverted_Exclamation
in Ada.Characters.Latin_1 A.3.3(21/3)
Inverted_Question
in Ada.Characters.Latin_1 A.3.3(22)
IS1 *in* Ada.Characters.Latin_1 A.3.3(16)
IS2 *in* Ada.Characters.Latin_1 A.3.3(16)
IS3 *in* Ada.Characters.Latin_1 A.3.3(16)
IS4 *in* Ada.Characters.Latin_1 A.3.3(16)
ISO_646_Set
in Ada.Strings.Maps.Constants A.4.6(4)
j
in Ada.Numerics.Generic_Complex_Types G.1.1(5)
in Interfaces.Fortran B.5(10)
Language_Unknown *in* Ada.Locales A.19(5/3)
LC_A *in* Ada.Characters.Latin_1 A.3.3(13)
LC_A_Acute *in* Ada.Characters.Latin_1 A.3.3(25)
LC_A_Circumflex
in Ada.Characters.Latin_1 A.3.3(25)
LC_A_Diaeresis
in Ada.Characters.Latin_1 A.3.3(25)
LC_A_Grave *in* Ada.Characters.Latin_1 A.3.3(25)

LC_A_Ring *in* Ada.Characters.Latin_1 A.3.3(25)
LC_A_Tilde *in* Ada.Characters.Latin_1 A.3.3(25)
LC_AE_Diphthong
in Ada.Characters.Latin_1 A.3.3(25)
LC_B *in* Ada.Characters.Latin_1 A.3.3(13)
LC_C *in* Ada.Characters.Latin_1 A.3.3(13)
LC_C_Cedilla
in Ada.Characters.Latin_1 A.3.3(25)
LC_D *in* Ada.Characters.Latin_1 A.3.3(13)
LC_E *in* Ada.Characters.Latin_1 A.3.3(13)
LC_E_Acute *in* Ada.Characters.Latin_1 A.3.3(25)
LC_E_Circumflex
in Ada.Characters.Latin_1 A.3.3(25)
LC_E_Diaeresis
in Ada.Characters.Latin_1 A.3.3(25)
LC_E_Grave *in* Ada.Characters.Latin_1 A.3.3(25)
LC_F *in* Ada.Characters.Latin_1 A.3.3(13)
LC_G *in* Ada.Characters.Latin_1 A.3.3(13)
LC_German_Sharp_S
in Ada.Characters.Latin_1 A.3.3(24)
LC_H *in* Ada.Characters.Latin_1 A.3.3(13)
LC_I *in* Ada.Characters.Latin_1 A.3.3(13)
LC_I_Acute *in* Ada.Characters.Latin_1 A.3.3(25)
LC_I_Circumflex
in Ada.Characters.Latin_1 A.3.3(25)
LC_I_Diaeresis
in Ada.Characters.Latin_1 A.3.3(25)
LC_I_Grave *in* Ada.Characters.Latin_1 A.3.3(25)
LC_Icelandic_Eth
in Ada.Characters.Latin_1 A.3.3(26)
LC_Icelandic_Thorn
in Ada.Characters.Latin_1 A.3.3(26)
LC_J *in* Ada.Characters.Latin_1 A.3.3(13)
LC_K *in* Ada.Characters.Latin_1 A.3.3(13)
LC_L *in* Ada.Characters.Latin_1 A.3.3(13)
LC_M *in* Ada.Characters.Latin_1 A.3.3(13)
LC_N *in* Ada.Characters.Latin_1 A.3.3(13)
LC_N_Tilde *in* Ada.Characters.Latin_1 A.3.3(26)
LC_O *in* Ada.Characters.Latin_1 A.3.3(13)
LC_O_Acute *in* Ada.Characters.Latin_1 A.3.3(26)
LC_O_Circumflex
in Ada.Characters.Latin_1 A.3.3(26)
LC_O_Diaeresis
in Ada.Characters.Latin_1 A.3.3(26)
LC_O_Grave *in* Ada.Characters.Latin_1 A.3.3(26)
LC_O_Oblique_Stroke
in Ada.Characters.Latin_1 A.3.3(26)
LC_O_Tilde *in* Ada.Characters.Latin_1 A.3.3(26)
LC_P *in* Ada.Characters.Latin_1 A.3.3(14)
LC_Q *in* Ada.Characters.Latin_1 A.3.3(14)
LC_R *in* Ada.Characters.Latin_1 A.3.3(14)
LC_S *in* Ada.Characters.Latin_1 A.3.3(14)
LC_T *in* Ada.Characters.Latin_1 A.3.3(14)
LC_U *in* Ada.Characters.Latin_1 A.3.3(14)
LC_U_Acute *in* Ada.Characters.Latin_1 A.3.3(26)
LC_U_Circumflex
in Ada.Characters.Latin_1 A.3.3(26)
LC_U_Diaeresis
in Ada.Characters.Latin_1 A.3.3(26)
LC_U_Grave *in* Ada.Characters.Latin_1 A.3.3(26)
LC_V *in* Ada.Characters.Latin_1 A.3.3(14)
LC_W *in* Ada.Characters.Latin_1 A.3.3(14)
LC_X *in* Ada.Characters.Latin_1 A.3.3(14)
LC_Y *in* Ada.Characters.Latin_1 A.3.3(14)
LC_Y_Acute *in* Ada.Characters.Latin_1 A.3.3(26)
LC_Y_Diaeresis
in Ada.Characters.Latin_1 A.3.3(26)
LC_Z *in* Ada.Characters.Latin_1 A.3.3(14)
Leading_Nonseparate
in Interfaces.COBOL B.4(23)
Leading_Separate *in* Interfaces.COBOL B.4(23)
Left_Angle_Quotation
in Ada.Characters.Latin_1 A.3.3(21/3)
Left_Curly_Bracket
in Ada.Characters.Latin_1 A.3.3(14)
Left_Parenthesis
in Ada.Characters.Latin_1 A.3.3(8)
Left_Square_Bracket
in Ada.Characters.Latin_1 A.3.3(12)
Less_Than_Sign
in Ada.Characters.Latin_1 A.3.3(10)
Letter_Set
in Ada.Strings.Maps.Constants A.4.6(4)
LF *in* Ada.Characters.Latin_1 A.3.3(5)
Low_Line *in* Ada.Characters.Latin_1 A.3.3(12)
Low_Order_First
in Interfaces.COBOL B.4(25)
in System 13.7(15/2)
Lower_Case_Map
in Ada.Strings.Maps.Constants A.4.6(5)
Lower_Set
in Ada.Strings.Maps.Constants A.4.6(4)
Macron *in* Ada.Characters.Latin_1 A.3.3(21/3)
Masculine_Ordinal_Indicator
in Ada.Characters.Latin_1 A.3.3(22)
Max_Base_Digits *in* System 13.7(8)
Max_Binary_Modulus *in* System 13.7(7)
Max_Decimal_Digits *in* Ada.Decimal F.2(5)
Max_Delta *in* Ada.Decimal F.2(4)
Max_Digits *in* System 13.7(8)
Max_Digits_Binary *in* Interfaces.COBOL B.4(11)
Max_Digits_Long_Binary
in Interfaces.COBOL B.4(11)
Max_Image_Width
in Ada.Numerics.Discrete_Random A.5.2(25)
in Ada.Numerics.Float_Random A.5.2(13)
Max_Int *in* System 13.7(6)
Max_Length *in* Ada.Strings.Bounded A.4.4(5)
Max_Mantissa *in* System 13.7(9)
Max_Nonbinary_Modulus *in* System 13.7(7)
Max_Picture_Length
in Ada.Text_IO.Editing F.3.3(8)
Max_Scale *in* Ada.Decimal F.2(3)
Memory_Size *in* System 13.7(13)
Micro_Sign *in* Ada.Characters.Latin_1 A.3.3(22)
Middle_Dot *in* Ada.Characters.Latin_1 A.3.3(22)
Min_Delta *in* Ada.Decimal F.2(4)
Min_Handler_Ceiling
in Ada.Execution_Time.Group_Budgets D.14.2(7/2)
in Ada.Execution_Time.Timers D.14.1(6/2)
Min_Int *in* System 13.7(6)
Min_Scale *in* Ada.Decimal F.2(3)
Minus_Sign *in* Ada.Characters.Latin_1 A.3.3(8)
Monday *in* Ada.Calendar.Formatting 9.6.1(17/2)
Multiplication_Sign
in Ada.Characters.Latin_1 A.3.3(24)
MW *in* Ada.Characters.Latin_1 A.3.3(18)
NAK *in* Ada.Characters.Latin_1 A.3.3(6)

Native_Binary *in* Interfaces.COBOL B.4(25)
NBH *in* Ada.Characters.Latin_1 A.3.3(17)
NBSP *in* Ada.Characters.Latin_1 A.3.3(21/3)
NEL *in* Ada.Characters.Latin_1 A.3.3(17)
New_Line_Count
in Ada.Strings.Text_Buffers A.4.12(5/5)
No_Break_Space
in Ada.Characters.Latin_1 A.3.3(21/3)
No_Element
in Ada.Containers.Doubly_Linked_Lists A.18.3(9/2), A.18.3(51.5/5)
in Ada.Containers.Hashed_Maps A.18.5(6/2), A.18.5(37.6/5)
in Ada.Containers.Hashed_Sets A.18.8(6/2), A.18.8(59.5/5)
in Ada.Containers.Multiway_Trees A.18.10(11/3), A.18.10(70.5/5)
in Ada.Containers.Ordered_Maps A.18.6(7/2), A.18.6(51.7/5)
in Ada.Containers.Ordered_Sets A.18.9(7/2), A.18.9(74.5/5)
in Ada.Containers.Vectors A.18.2(11/2), A.18.2(79.5/5)
No_Index *in* Ada.Containers.Vectors A.18.2(7/2)
No_Tag *in* Ada.Tags 3.9(6.1/2)
Not_A_Specific_CPU
in System.Multiprocessors D.16(4/3)
Not_Sign *in* Ada.Characters.Latin_1 A.3.3(21/3)
NUL
in Ada.Characters.Latin_1 A.3.3(5)
in Interfaces.C B.3(20/1)
Null_Address *in* System 13.7(12)
Null_Bounded_String
in Ada.Strings.Bounded A.4.4(7)
Null_Id *in* Ada.Exceptions 11.4.1(2/5)
Null_Occurrence *in* Ada.Exceptions 11.4.1(3/5)
Null_Ptr *in* Interfaces.C.Strings B.3.1(7)
Null_Set
in Ada.Strings.Maps A.4.2(5)
in Ada.Strings.Wide_Maps A.4.7(5)
in Ada.Strings.Wide_Wide_Maps A.4.8(5/2)
Null_Unbounded_String
in Ada.Strings.Unbounded A.4.5(5)
Number_Sign *in* Ada.Characters.Latin_1 A.3.3(8)
OSC *in* Ada.Characters.Latin_1 A.3.3(19)
Packed_Signed *in* Interfaces.COBOL B.4(27)
Packed_Unsigned *in* Interfaces.COBOL B.4(27)
Paragraph_Sign
in Ada.Characters.Latin_1 A.3.3(22)
Percent_Sign
in Ada.Characters.Latin_1 A.3.3(8)
Pi *in* Ada.Numerics A.5(3/5)
Pilcrow_Sign
in Ada.Characters.Latin_1 A.3.3(22)
PLD *in* Ada.Characters.Latin_1 A.3.3(17)
PLU *in* Ada.Characters.Latin_1 A.3.3(17)
Plus_Minus_Sign
in Ada.Characters.Latin_1 A.3.3(22)
Plus_Sign *in* Ada.Characters.Latin_1 A.3.3(8)
PM *in* Ada.Characters.Latin_1 A.3.3(19)
Pound_Sign *in* Ada.Characters.Latin_1 A.3.3(21/3)
PU1 *in* Ada.Characters.Latin_1 A.3.3(18)
PU2 *in* Ada.Characters.Latin_1 A.3.3(18)
Question *in* Ada.Characters.Latin_1 A.3.3(10)
Quotation *in* Ada.Characters.Latin_1 A.3.3(8)
Registered_Trade_Mark_Sign
in Ada.Characters.Latin_1 A.3.3(21/3)
Reserved_128
in Ada.Characters.Latin_1 A.3.3(17)
Reserved_129
in Ada.Characters.Latin_1 A.3.3(17)
Reserved_132
in Ada.Characters.Latin_1 A.3.3(17)
Reserved_153
in Ada.Characters.Latin_1 A.3.3(19)
Reverse_Solidus
in Ada.Characters.Latin_1 A.3.3(12)
RI *in* Ada.Characters.Latin_1 A.3.3(17)
Right_Angle_Quotation
in Ada.Characters.Latin_1 A.3.3(22)
Right_Curly_Bracket
in Ada.Characters.Latin_1 A.3.3(14)
Right_Parenthesis
in Ada.Characters.Latin_1 A.3.3(8)
Right_Square_Bracket
in Ada.Characters.Latin_1 A.3.3(12)
Ring_Above *in* Ada.Characters.Latin_1 A.3.3(22)
RS *in* Ada.Characters.Latin_1 A.3.3(6)
Saturday *in* Ada.Calendar.Formatting 9.6.1(17/2)
SCHAR_MAX *in* Interfaces.C B.3(6)
SCHAR_MIN *in* Interfaces.C B.3(6)
SCI *in* Ada.Characters.Latin_1 A.3.3(19)
Section_Sign
in Ada.Characters.Latin_1 A.3.3(21/3)
Semicolon *in* Ada.Characters.Latin_1 A.3.3(10)
Separate_Interrupt_Clocks_Supported
in Ada.Execution_Time D.14(9.2/3)
SI *in* Ada.Characters.Latin_1 A.3.3(5)
SO *in* Ada.Characters.Latin_1 A.3.3(5)
Soft_Hyphen *in* Ada.Characters.Latin_1 A.3.3(21/3)
SOH *in* Ada.Characters.Latin_1 A.3.3(5)
Solidus *in* Ada.Characters.Latin_1 A.3.3(8)
SOS *in* Ada.Characters.Latin_1 A.3.3(19)
SPA *in* Ada.Characters.Latin_1 A.3.3(18)
Space
in Ada.Characters.Latin_1 A.3.3(8)
in Ada.Strings A.4.1(4/2)
Special_Set
in Ada.Strings.Maps.Constants A.4.6(4)
SS2 *in* Ada.Characters.Latin_1 A.3.3(17)
SS3 *in* Ada.Characters.Latin_1 A.3.3(17)
SSA *in* Ada.Characters.Latin_1 A.3.3(17)
ST *in* Ada.Characters.Latin_1 A.3.3(19)
Standard_Indent
in Ada.Strings.Text_Buffers A.4.12(13/5)
Storage_Unit *in* System 13.7(13)
STS *in* Ada.Characters.Latin_1 A.3.3(18)
STX *in* Ada.Characters.Latin_1 A.3.3(5)
SUB *in* Ada.Characters.Latin_1 A.3.3(6)
Success *in* Ada.Command_Line A.15(8)
Sunday *in* Ada.Calendar.Formatting 9.6.1(17/2)
Superscript_One
in Ada.Characters.Latin_1 A.3.3(22)
Superscript_Three
in Ada.Characters.Latin_1 A.3.3(22)
Superscript_Two
in Ada.Characters.Latin_1 A.3.3(22)
SYN *in* Ada.Characters.Latin_1 A.3.3(6)
System_Dispatching_Domain
in System.Multiprocessors.Dispatching_Domains D.16.1(6/3)

System_Name *in* System 13.7(4)
Thursday *in* Ada.Calendar.Formatting 9.6.1(17/2)
Tick
in Ada.Real_Time D.8(6)
in System 13.7(10)
Tilde *in* Ada.Characters.Latin_1 A.3.3(14)
Time_First *in* Ada.Real_Time D.8(4)
Time_Last *in* Ada.Real_Time D.8(4)
Time_Span_First *in* Ada.Real_Time D.8(5)
Time_Span_Last *in* Ada.Real_Time D.8(5)
Time_Span_Unit *in* Ada.Real_Time D.8(5)
Time_Span_Zero *in* Ada.Real_Time D.8(5)
Time_Unit *in* Ada.Real_Time D.8(4)
Trailing_Nonseparate
in Interfaces.COBOL B.4(23)
Trailing_Separate *in* Interfaces.COBOL B.4(23)
Tuesday *in* Ada.Calendar.Formatting 9.6.1(17/2)
UC_A_Acute *in* Ada.Characters.Latin_1 A.3.3(23)
UC_A_Circumflex
in Ada.Characters.Latin_1 A.3.3(23)
UC_A_Diaeresis
in Ada.Characters.Latin_1 A.3.3(23)
UC_A_Grave *in* Ada.Characters.Latin_1 A.3.3(23)
UC_A_Ring *in* Ada.Characters.Latin_1 A.3.3(23)
UC_A_Tilde *in* Ada.Characters.Latin_1 A.3.3(23)
UC_AE_Diphthong
in Ada.Characters.Latin_1 A.3.3(23)
UC_C_Cedilla
in Ada.Characters.Latin_1 A.3.3(23)
UC_E_Acute *in* Ada.Characters.Latin_1 A.3.3(23)
UC_E_Circumflex
in Ada.Characters.Latin_1 A.3.3(23)
UC_E_Diaeresis
in Ada.Characters.Latin_1 A.3.3(23)
UC_E_Grave *in* Ada.Characters.Latin_1 A.3.3(23)
UC_I_Acute *in* Ada.Characters.Latin_1 A.3.3(23)
UC_I_Circumflex
in Ada.Characters.Latin_1 A.3.3(23)
UC_I_Diaeresis
in Ada.Characters.Latin_1 A.3.3(23)
UC_I_Grave *in* Ada.Characters.Latin_1 A.3.3(23)
UC_Icelandic_Eth
in Ada.Characters.Latin_1 A.3.3(24)
UC_Icelandic_Thorn
in Ada.Characters.Latin_1 A.3.3(24)
UC_N_Tilde *in* Ada.Characters.Latin_1 A.3.3(24)
UC_O_Acute *in* Ada.Characters.Latin_1 A.3.3(24)
UC_O_Circumflex
in Ada.Characters.Latin_1 A.3.3(24)
UC_O_Diaeresis
in Ada.Characters.Latin_1 A.3.3(24)
UC_O_Grave *in* Ada.Characters.Latin_1 A.3.3(24)
UC_O_Oblique_Stroke
in Ada.Characters.Latin_1 A.3.3(24)
UC_O_Tilde *in* Ada.Characters.Latin_1 A.3.3(24)
UC_U_Acute *in* Ada.Characters.Latin_1 A.3.3(24)
UC_U_Circumflex
in Ada.Characters.Latin_1 A.3.3(24)
UC_U_Diaeresis
in Ada.Characters.Latin_1 A.3.3(24)
UC_U_Grave *in* Ada.Characters.Latin_1 A.3.3(24)
UC_Y_Acute *in* Ada.Characters.Latin_1 A.3.3(24)
UCHAR_MAX *in* Interfaces.C B.3(6)
Unbounded *in* Ada.Text_IO A.10.1(5)
Unsigned *in* Interfaces.COBOL B.4(23)
Upper_Case_Map
in Ada.Strings.Maps.Constants A.4.6(5)
Upper_Set
in Ada.Strings.Maps.Constants A.4.6(4)
US *in* Ada.Characters.Latin_1 A.3.3(6)
Vertical_Line
in Ada.Characters.Latin_1 A.3.3(14)
VT *in* Ada.Characters.Latin_1 A.3.3(5)
VTS *in* Ada.Characters.Latin_1 A.3.3(17)
Wednesday *in* Ada.Calendar.Formatting 9.6.1(17/2)
Wide_Character_Set
in Ada.Strings.Wide_Maps.Wide_Constants A.4.8(48/2)
wide_nul *in* Interfaces.C B.3(31/1)
Wide_Space *in* Ada.Strings A.4.1(4/2)
Wide_Wide_Space *in* Ada.Strings A.4.1(4/2)
Word_Size *in* System 13.7(13)
Yen_Sign *in* Ada.Characters.Latin_1 A.3.3(21/3)

Index

Index entries are given by paragraph number.

& operator 4.5(1), 4.5.3(3)

* operator 4.5(1), 4.5.5(1)
** operator 4.5(1), 4.5.6(7)

+ operator 4.5(1), 4.5.3(1), 4.5.4(1)

- operator 4.5(1), 4.5.3(1), 4.5.4(1)

/ operator 4.5(1), 4.5.5(1)
/= operator 4.5(1), 4.5.2(1), 6.6(6/3)

10646:2020, ISO/IEC standard 1.2(8/5)
14882:2020, ISO/IEC standard 1.2.1(9/5)
1539-1:2018, ISO/IEC standard 1.2.1(3/5)
1989:2002, ISO standard 1.2.1(4/5)

3166-1:2020, ISO standard 1.2(4.1/5)

60559:2020, ISO/IEC standard 1.2.1(10/5)
639-3:2007, ISO standard 1.2(1.1/3)
6429:1992, ISO/IEC standard 1.2.1(5/5)
646:1991, ISO/IEC standard 1.2.1(2/5)

8601-1:2019, ISO standard 1.2.1(6/5)
8859-1:1998, ISO/IEC standard 1.2.1(7/5)

9899:2018, ISO/IEC standard 1.2.1(8/5)

< operator 4.5(1), 4.5.2(1)
<= operator 4.5(1), 4.5.2(1)

= operator 4.5(1), 4.5.2(1)

> operator 4.5(1), 4.5.2(1)
>= operator 4.5(1), 4.5.2(1)

A

abnormal completion 7.6.1(2/2)
abnormal state of an object 13.9.1(4)
  [*partial*] 9.8(21), 11.6(6/5), A.13(17)
abnormal task 9.8(4)
abnormal termination
  of a partition 10.2(25.c)
abort
  of a partition E.1(7)
  of a task 9.8(4)
  of the execution of a construct 9.8(5)
abort completion point 9.8(15)
abort-deferred operation 9.8(5)
abort_statement 9.8(2)
  used 5.1(4/2), P.1
Abort_Task
  in Ada.Task_Identification C.7.1(3/5)
abortable_part 9.7.4(5)
  used 9.7.4(2), P.1
abs operator 4.5(1), 4.5.6(1)
absolute deadline
  active D.2.6(14/5)
  base D.2.6(14/5)
absolute value 4.5(1), 4.5.6(1)
abstract data type (ADT)
  See private types and private extensions 7.3(1)
  See also abstract type 3.9.3(1/2)
abstract formal subprogram 12.6(8.d/2)
abstract subprogram 3.9.3(1/2), 3.9.3(3/2)
abstract type 1.3.1(1/5), 3.9.3(1.2/2), 3.9.3(1/2)
abstract_subprogram_declaration 3.9.3(1.1/3)
  used 3.1(3/3), P.1
ACATS
  Ada Conformity Assessment Test Suite 1.3(1.f/5)
accept_alternative 9.7.1(5)
  used 9.7.1(4), P.1
accept_statement 9.5.2(3)
  used 5.1(5/5), 9.7.1(5), P.1
acceptable interpretation 8.6(14)
Access attribute 3.10.2(24/1), 3.10.2(32/5)
  See also Unchecked_Access attribute 13.10(3)
access discriminant 3.7(9/2)
access parameter 6.1(24/2)
access paths
  distinct 6.2(12/3)
access result type 6.1(24/2)
access type 1.3.1(2/5), 3.10(1)
  subpool 13.11.4(22/3)
access types
  input-output unspecified A.7(6)
access value 3.10(1)
access-to-constant type 3.10(10)
access-to-object type 3.10(7/1)
access-to-subprogram type 3.10(7/1), 3.10(11)
access-to-variable type 3.10(10)
Access_Check 11.5(11/2)
  [*partial*] 4.1(13), 4.1.5(8/3), 4.6(51/5), 4.8(10.4/3)
access_definition 3.10(6/2)
  used 3.3.1(2/3), 3.6(7/2), 3.7(5/5), 5.5.2(2.1/5), 6.1(13/2), 6.1(15/5), 6.5(2.3/2), 8.5.1(2/5), 12.4(2/3), P.1
access_to_object_definition 3.10(3)
  used 3.10(2/2), P.1
access_to_subprogram_definition 3.10(5)
  used 3.10(2/2), P.1
access_type_definition 3.10(2/2)
  used 3.2.1(4/2), 12.5.4(2), P.1
accessibility
  distributed 3.10.2(32.1/3)
  from shared passive library units E.2.1(8)
accessibility level 1.3.1(4/5), 3.10.2(3/5)
accessibility rule
  Access attribute 3.10.2(28/3), 3.10.2(32/5)
  checking in generic units 12.3(11.s)
  not part of generic contract 3.9.1(4.k)
  requeue statement 9.5.4(6/3)
  type conversion 4.6(24.17/4), 4.6(24.21/4)
  type conversion, array components 4.6(24.6/2)
accessibility rules
  See also Heart of Darkness 3.10.2(3.b/3)
Accessibility_Check 11.5(19.1/2)
  [*partial*] 3.10.2(29), 4.6(39.1/2), 4.6(48/3), 4.8(10.1/3), 6.5(8/5), 6.5(21/3), 9.5.4(7.2/5), 13.11.4(25/3), 13.11.4(26/3), E.4(18/1)
accessible partition E.1(7)
accumulator
  reduction attribute 4.5.10(24/5)
accuracy 4.6(32), G.2(1)
ACID 1.3(1.f/5)
ACK
  in Ada.Characters.Latin_1 A.3.3(5)
acquire
  execution resource associated with protected object 9.5.1(5/4)
actions
  concurrent 9.10(16/5)
  conflicting 9.10(17/5)
  potentially conflicting 9.10(17/5)
  sequential 9.10(11/5)
activation
  of a task 9.2(1)
activation failure 9.2(1)
Activation_Is_Complete
  in Ada.Task_Identification C.7.1(4/3)
activator
  of a task 9.2(5)
active
  absolute deadline D.2.6(14/5)
active locale A.19(8/3)
active partition 10.2(28/3), E.1(2)
active priority D.1(15/5)
actual 12.3(7/5)
actual duration D.9(12/5)

actual parameter
  for a formal parameter 6.4.1(3)
actual subtype 3.3(23/5), 12.5(4)
  of an object 3.3.1(9/2)
actual type 12.5(4)
actual_parameter_part 6.4(4)
  used 4.1.6(10/3), 6.4(2), 6.4(3), 9.5.3(2), P.1
Actual_Quantum
  in Ada.Dispatching.Round_Robin D.2.5(4/5)
Acute
  in Ada.Characters.Latin_1 A.3.3(22)
ACVC
  Ada Compiler Validation Capability 1.3(1.f/5)
Ada A.2(2/5)
Ada calling convention 6.3.1(3/3)
Ada Commentary Integration Document (ACID) 1.3(1.f/5)
Ada Compiler Validation Capability ACVC 1.3(1.f/5)
Ada Conformity Assessment Test Suite ACATS 1.3(1.f/5)
Ada Issue (AI) 1.3(1.f/5)
Ada Rapporteur Group (ARG) 1.3(1.f/5)
Ada.Ada.Unchecked_Deallocate_Subpool 13.11.5(3/5)
Ada.Assertions 11.4.2(12/5)
Ada.Asynchronous_Task_Control D.11(3/5)
Ada.Calendar 9.6(10/5)
Ada.Calendar.Arithmetic 9.6.1(8/5)
Ada.Calendar.Formatting 9.6.1(15/5)
Ada.Calendar.Time_Zones 9.6.1(2/5)
Ada.Characters A.3.1(2/5)
Ada.Characters.Conversions A.3.4(2/5)
Ada.Characters.Handling A.3.2(2/5)
Ada.Characters.Latin_1 A.3.3(3/5)
Ada.Command_Line A.15(3/5)
Ada.Complex_Text_IO G.1.3(9.1/2)
Ada.Containers A.18.1(3/5)
Ada.Containers.Bounded_Doubly_Linked_Lists A.18.20(1/3)
Ada.Containers.Bounded_Hashed_Maps A.18.21(1/3)
Ada.Containers.Bounded_Hashed_Sets A.18.23(1/3)
Ada.Containers.Bounded_Indefinite_Holders A.18.32(1/5)
Ada.Containers.Bounded_Multiway_Trees A.18.25(1/3)
Ada.Containers.Bounded_Ordered_Maps A.18.22(1/3)
Ada.Containers.Bounded_Ordered_Sets A.18.24(1/3)
Ada.Containers.Bounded_Priority_Queues A.18.31(2/5)
Ada.Containers.Bounded_Synchronized_Queues A.18.29(2/5)
Ada.Containers.Bounded_Vectors A.18.19(1/3)
Ada.Containers.Doubly_Linked_Lists A.18.3(5/5)
Ada.Containers.Generic_Array_Sort A.18.26(3/5)
Ada.Containers.Generic_Constrained_Array_Sort A.18.26(7/5)
Ada.Containers.Generic_Sort A.18.26(9.2/5)
Ada.Containers.Hashed_Maps A.18.5(2/5)
Ada.Containers.Hashed_Sets A.18.8(2/5)
Ada.Containers.Indefinite_Doubly_Linked_Lists A.18.12(2/3)
Ada.Containers.Indefinite_Hashed_Maps A.18.13(2/3)
Ada.Containers.Indefinite_Hashed_Sets A.18.15(2/3)
Ada.Containers.Indefinite_Holders A.18.18(5/5)
Ada.Containers.Indefinite_Multiway_Trees A.18.17(2/3)
Ada.Containers.Indefinite_Ordered_Maps A.18.14(2/3)
Ada.Containers.Indefinite_Ordered_Sets A.18.16(2/3)
Ada.Containers.Indefinite_Vectors A.18.11(2/3)
Ada.Containers.Multiway_Trees A.18.10(7/5)
Ada.Containers.Ordered_Maps A.18.6(2/5)
Ada.Containers.Ordered_Sets A.18.9(2/5)
Ada.Containers.Synchronized_Queue_Interfaces A.18.27(3/5)
Ada.Containers.Unbounded_Priority_Queues A.18.30(2/5)
Ada.Containers.Unbounded_Synchronized_Queues A.18.28(2/5)
Ada.Containers.Vectors A.18.2(6/5)
Ada.Decimal F.2(2/5)
Ada.Direct_IO A.8.4(2/5)
Ada.Directories A.16(3/5)
Ada.Directories.Hierarchical_File_Names A.16.1(3/5)
Ada.Directories.Information A.16(124/2)
Ada.Dispatching D.2.1(1.2/5)
Ada.Dispatching.EDF D.2.6(9/5)
Ada.Dispatching.Non_Preemptive D.2.4(2.2/5)
Ada.Dispatching.Round_Robin D.2.5(4/5)
Ada.Dynamic_Priorities D.5.1(3/5)
Ada.Environment_Variables A.17(3/5)
Ada.Exceptions 11.4.1(2/5)
Ada.Execution_Time D.14(3/5)
Ada.Execution_Time.Group_Budgets D.14.2(3/5)
Ada.Execution_Time.Interrupts D.14.3(3/5)
Ada.Execution_Time.Timers D.14.1(3/5)
Ada.Finalization 7.6(4/5)
Ada.Float_Text_IO A.10.9(33)
Ada.Float_Wide_Text_IO A.11(2/2)
Ada.Float_Wide_Wide_Text_IO A.11(3/2)
Ada.Integer_Text_IO A.10.8(21)
Ada.Integer_Wide_Text_IO A.11(2/2)
Ada.Integer_Wide_Wide_Text_IO A.11(3/2)
Ada.Interrupts C.3.2(2/5)
Ada.Interrupts.Names C.3.2(12/5)
Ada.IO_Exceptions A.13(3/5)
Ada.Iterator_Interfaces 5.5.1(2/5)
Ada.Locales A.19(3/5)
Ada.Numerics A.5(3/5)
Ada.Numerics.Big_Numbers.Big_Integers A.5.5(3/5), A.5.6(2/5)
Ada.Numerics.Big_Numbers.Big_Reals A.5.7(2/5)
Ada.Numerics.Complex_Arrays G.3.2(53/2)
Ada.Numerics.Complex_Elementary_Functions G.1.2(9/1)
Ada.Numerics.Complex_Types G.1.1(25/1)
Ada.Numerics.Discrete_Random A.5.2(17/5)
Ada.Numerics.Elementary_Functions A.5.1(9/1)
Ada.Numerics.Float_Random A.5.2(5/5)
Ada.Numerics.Generic_Complex_Arrays G.3.2(2/5)
Ada.Numerics.Generic_Complex_Elementary_Functions G.1.2(2/5)
Ada.Numerics.Generic_Complex_Types G.1.1(2/5)
Ada.Numerics.Generic_Elementary_Functions A.5.1(3/5)
Ada.Numerics.Generic_Real_Arrays G.3.1(2/5)
Ada.Numerics.Real_Arrays G.3.1(31/2)
Ada.Real_Time D.8(3/5)
Ada.Real_Time.Timing_Events D.15(3/5)
Ada.Sequential_IO A.8.1(2/5)
Ada.Storage_IO A.9(3/5)
Ada.Streams 13.13.1(2/5)
Ada.Streams.Storage 13.13.1(12/5)
Ada.Streams.Storage.Bounded 13.13.1(26/5)
Ada.Streams.Storage.Unbounded 13.13.1(18/5)
Ada.Streams.Stream_IO A.12.1(3/5)
Ada.Strings A.4.1(3/5)
Ada.Strings.Bounded A.4.4(3/5)
Ada.Strings.Bounded.Equal_Case_Insensitive A.4.10(7/5)
Ada.Strings.Bounded.Hash A.4.9(7/5)
Ada.Strings.Bounded.Hash_Case_Insensitive A.4.9(11.7/5)
Ada.Strings.Bounded.Less_Case_Insensitive A.4.10(18/5)
Ada.Strings.Equal_Case_Insensitive A.4.10(2/5)
Ada.Strings.Fixed A.4.3(5/5)
Ada.Strings.Fixed.Equal_Case_Insensitive A.4.10(5/3)
Ada.Strings.Fixed.Hash_Case_Insensitive A.4.9(11.5/3)

Ada.Strings.Fixed.Less_Case_Insensitive A.4.10(16/3)
Ada.Strings.Hash A.4.9(2/5)
Ada.Strings.Hash_Case_Insensitive A.4.9(11.2/5)
Ada.Strings.Less_Case_Insensitive A.4.10(13/5)
Ada.Strings.Maps A.4.2(3/5)
Ada.Strings.Maps.Constants A.4.6(3/5)
Ada.Strings.Text_Buffers A.4.12(3/5)
Ada.Strings.Text_Buffers.Bounded A.4.12(26/5)
Ada.Strings.Text_Buffers.Unbounded A.4.12(18/5)
Ada.Strings.Unbounded A.4.5(3/5)
Ada.Strings.Unbounded.Equal_Case_Insensitive A.4.10(10/5)
Ada.Strings.Unbounded.Hash A.4.9(10/5)
Ada.Strings.Unbounded.Hash_Case_Insensitive A.4.9(11.10/5)
Ada.Strings.Unbounded.Less_Case_Insensitive A.4.10(21/5)
Ada.Strings.UTF_Encoding A.4.11(3/5)
Ada.Strings.UTF_Encoding.Conversions A.4.11(15/5)
Ada.Strings.UTF_Encoding.Strings A.4.11(22/5)
Ada.Strings.UTF_Encoding.Wide_Strings A.4.11(30/5)
Ada.Strings.UTF_Encoding.Wide_Wide_Strings A.4.11(38/5)
Ada.Strings.Wide_Bounded A.4.7(1/3)
Ada.Strings.Wide_Bounded.Wide_Equal_Case_Insensitive A.4.7(1/3)
Ada.Strings.Wide_Bounded.Wide_Hash A.4.7(1/3)
Ada.Strings.Wide_Bounded.Wide_Hash_Case_Insensitive A.4.7(1/3)
Ada.Strings.Wide_Equal_Case_Insensitive A.4.7(1/3)
Ada.Strings.Wide_Fixed A.4.7(1/3)
Ada.Strings.Wide_Fixed.Wide_Equal_Case_Insensitive A.4.7(1/3)
Ada.Strings.Wide_Fixed.Wide_Hash A.4.7(1/3)
Ada.Strings.Wide_Fixed.Wide_Hash_Case_Insensitive A.4.7(1/3)
Ada.Strings.Wide_Hash A.4.7(1/3)
Ada.Strings.Wide_Hash_Case_Insensitive A.4.7(1/3)
Ada.Strings.Wide_Maps A.4.7(3/5)
Ada.Strings.Wide_Maps.Wide_Constants A.4.7(1/3), A.4.8(28/2)
Ada.Strings.Wide_Unbounded A.4.7(1/3)
Ada.Strings.Wide_Unbounded.Wide_Equal_Case_Insensitive A.4.7(1/3)
Ada.Strings.Wide_Unbounded.Wide_Hash A.4.7(1/3)
Ada.Strings.Wide_Unbounded.Wide_Hash_Case_Insensitive A.4.7(1/3)
Ada.Strings.Wide_Wide_Bounded A.4.8(1/3)
Ada.Strings.Wide_Wide_Bounded.Wide_Wide_Equal_Case_Insensitive A.4.8(1/3)
Ada.Strings.Wide_Wide_Bounded.Wide_Wide_Hash A.4.8(1/3)
Ada.Strings.Wide_Wide_Bounded.Wide_Wide_Hash_Case_Insensitive A.4.8(1/3)
Ada.Strings.Wide_Wide_Equal_Case_Insensitive A.4.8(1/3)
Ada.Strings.Wide_Wide_Fixed A.4.8(1/3)
Ada.Strings.Wide_Wide_Fixed.Wide_Wide_Equal_Case_Insensitive A.4.8(1/3)
Ada.Strings.Wide_Wide_Fixed.Wide_Wide_Hash A.4.8(1/3)
Ada.Strings.Wide_Wide_Fixed.Wide_Wide_Hash_Case_Insensitive A.4.8(1/3)
Ada.Strings.Wide_Wide_Hash A.4.8(1/3)
Ada.Strings.Wide_Wide_Hash_Case_Insensitive A.4.8(1/3)
Ada.Strings.Wide_Wide_Maps A.4.8(3/5)
Ada.Strings.Wide_Wide_Maps.Wide_Wide_Constants A.4.8(1/3)
Ada.Strings.Wide_Wide_Unbounded A.4.8(1/3)
Ada.Strings.Wide_Wide_Unbounded.Wide_Wide_Equal_Case_Insensitive A.4.8(1/3)
Ada.Strings.Wide_Wide_Unbounded.Wide_Wide_Hash A.4.8(1/3)
Ada.Strings.Wide_Wide_Unbounded.Wide_Wide_Hash_Case_Insensitive A.4.8(1/3)
Ada.Synchronous_Barriers D.10.1(3/5)
Ada.Synchronous_Task_Control D.10(3/5)
Ada.Synchronous_Task_Control.EDF D.10(5.2/5)
Ada.Tags 3.9(6/5)
Ada.Tags.Generic_Dispatching_Constructor 3.9(18.2/5)
Ada.Task_Attributes C.7.2(2/5)
Ada.Task_Identification C.7.1(2/5)
Ada.Task_Termination C.7.3(2/5)
Ada.Text_IO A.10.1(2/5)
Ada.Text_IO.Bounded_IO A.10.11(3/5)
Ada.Text_IO.Complex_IO G.1.3(3/5)
Ada.Text_IO.Editing F.3.3(3/5)
Ada.Text_IO.Text_Streams A.12.2(3/5)
Ada.Text_IO.Unbounded_IO A.10.12(3/5)
Ada.Unchecked_Conversion 13.9(3/5)
Ada.Unchecked_Deallocate_Subpool
  child of Ada 13.11.5(3/5)
Ada.Unchecked_Deallocation 13.11.2(3/5)
Ada.Wide_Characters A.3.1(4/5)
Ada.Wide_Characters.Handling A.3.5(3/5)
Ada.Wide_Command_Line A.15.1(2/5)
Ada.Wide_Directories A.16.2(2/5)
Ada.Wide_Directories.Hierarchical_File_Names A.16.2(2/5)
Ada.Wide_Directories.Information A.16.2(2/5)
Ada.Wide_Environment_Variables A.17.1(2/5)
Ada.Wide_Text_IO A.11(2/2)
Ada.Wide_Text_IO.Complex_IO G.1.4(1)
Ada.Wide_Text_IO.Editing F.3.4(1)
Ada.Wide_Text_IO.Text_Streams A.12.3(3/5)
Ada.Wide_Text_IO.Wide_Bounded_IO A.11(4/3)
Ada.Wide_Text_IO.Wide_Unbounded_IO A.11(5/3)
Ada.Wide_Wide_Characters A.3.1(6/5)
Ada.Wide_Wide_Characters.Handling A.3.6(1/3)
Ada.Wide_Wide_Command_Line A.15.1(3/5)
Ada.Wide_Wide_Directories A.16.2(3/5)
Ada.Wide_Wide_Directories.Hierarchical_File_Names A.16.2(3/5)
Ada.Wide_Wide_Directories.Information A.16.2(3/5)
Ada.Wide_Wide_Environment_Variables A.17.1(3/5)
Ada.Wide_Wide_Text_IO A.11(3/2)
Ada.Wide_Wide_Text_IO.Complex_IO G.1.5(1/2)
Ada.Wide_Wide_Text_IO.Editing F.3.5(1/2)
Ada.Wide_Wide_Text_IO.Text_Streams A.12.4(3/5)
Ada.Wide_Wide_Text_IO.Wide_Wide_Bounded_IO A.11(4/3)
Ada.Wide_Wide_Text_IO.Wide_Wide_Unbounded_IO A.11(5/3)
Ada_To_COBOL
  in Interfaces.COBOL B.4(14)
adafinal B.1(39/3)
adainit B.1(39/3)
Add
  in Ada.Execution_Time.Group_Budgets D.14.2(9/2)
Add_Task
  in Ada.Execution_Time.Group_Budgets D.14.2(8/2)
additive
  aspect 13.1.1(18.3/5)
address
  arithmetic 13.7.1(6)
  comparison 13.7(14/3)
  in System 13.7(12)

Address aspect 13.3(12)
Address attribute 13.3(11), J.7.1(5)
Address clause 13.3(7/2), 13.3(12)
Address_To_Access_Conversions
  child of System 13.7.2(2/5)
Adjacent attribute A.5.3(48)
Adjust 7.6(2)
  in Ada.Finalization 7.6(6/2)
adjusting the value of an object 7.6(15), 7.6(16/3)
adjustment 7.6(15), 7.6(16/3)
  as part of assignment 5.2(14/5)
Admission_Policy pragma D.4.1(3/5), L(1.1/5)
ADT (abstract data type)
  See private types and private extensions 7.3(1)
  See also abstract type 3.9.3(1/2)
advice 1.1.2(37)
Aft attribute 3.5.10(5)
aggregate 1.3.3(1/5), 4.3(1), 4.3(2/5)
  index parameter 4.3.3(6.1/5)
  used 4.4(7/5), 4.7(2), 6.8(2/4), 13.1.1(4/5), P.1
  See also composite type 3.2(2/5)
Aggregate aspect 4.3.5(3/5)
AI 1.3(1.f/5)
aliased 3.10(9/5)
aliased view 1.3.1(5/5)
aliasing
  See distinct access paths 6.2(12/3)
Alignment
  in Ada.Strings A.4.1(6)
Alignment (subtype) aspect 13.3(26.4/2)
Alignment attribute 13.3(23/2), 13.3(26.2/2)
Alignment clause 13.3(7/2), 13.3(25/2), 13.3(26.4/2)
All_Calls_Remote aspect E.2.3(16/5)
All_Calls_Remote pragma J.15.15(6/5), L(2.1/5)
All_Checks 11.5(25/3)
All_Conflict_Checks conflict check policy 9.10.1(22/5)
All_Parallel_Conflict_Checks conflict check policy 9.10.1(20/5)
All_Tasking_Conflict_Checks conflict check policy 9.10.1(21/5)
Allocate
  in System.Storage_Pools 13.11(7)
  in System.Storage_Pools.Subpools 13.11.4(14/3)
Allocate_From_Subpool
  in System.Storage_Pools.Subpools 13.11.4(11/5)
Allocation_Check 11.5(19.2/2)
  [*partial*] 4.8(10.2/2), 4.8(10.3/2), 4.8(10.4/3), 13.11.4(30/3)
allocator 4.8(2/3)
  used 4.4(7/5), P.1
allows blocking 9.5(21/5)
allows exit 5.5.3(19/5)
Alphanumeric
  in Interfaces.COBOL B.4(16/3)
alphanumeric character
  a category of Character A.3.2(31)
Alphanumeric_Set
  in Ada.Strings.Maps.Constants A.4.6(4)
ambiguous 8.6(30)
ambiguous cursor
  of a vector A.18.2(240/2)
ambiguous grammar 1.1.4(14.a)
Amendment Correction 1.1.2(39.n/3)
ampersand 2.1(15/5)
  in Ada.Characters.Latin_1 A.3.3(8)
ampersand operator 4.5(1), 4.5.3(3)
ancestor
  of a library unit 10.1.1(11)
  of a tree node A.18.10(4/3)
  of a type 3.4.1(10/2)
  ultimate 3.4.1(10/2)
ancestor of a type 1.3.1(7/5)
ancestor subtype
  of a formal derived type 12.5.1(5/3)
  of a private_extension_declaration 7.3(8)
ancestor type
  of an extension_aggregate 4.3.2(5/3)
Ancestor_Find
  in Ada.Containers.Multiway_Trees A.18.10(40.1/5), A.18.10(40/5)
ancestor_part 4.3.2(3)
  used 4.3.2(2), P.1
and operator 4.5(1), 4.5.1(2)
and then (short-circuit control form) 4.5(1), 4.5.1(1)
angle threshold G.2.4(10)
Annex
  informative 1.1.2(18)
  normative 1.1.2(14)
  Specialized Needs 1.1.2(7)
anonymous access type 3.10(12/3)
anonymous allocator 3.10.2(14/3)
anonymous array type 3.3.1(1/3)
anonymous protected type 3.3.1(1/3)
anonymous task type 3.3.1(1/3)
anonymous type 3.2.1(7/2)
Any_Priority *subtype of* Integer
  in System 13.7(16)
APC
  in Ada.Characters.Latin_1 A.3.3(19)
apostrophe 2.1(15/5)
  in Ada.Characters.Latin_1 A.3.3(8)
Append
  in Ada.Containers.Doubly_Linked_Lists A.18.3(23.1/5), A.18.3(23/5)
  in Ada.Containers.Vectors A.18.2(47.1/5), A.18.2(47/5)
  in Ada.Strings.Bounded A.4.4(13), A.4.4(14), A.4.4(15), A.4.4(16), A.4.4(17), A.4.4(18), A.4.4(19), A.4.4(20)
  in Ada.Strings.Unbounded A.4.5(12), A.4.5(13), A.4.5(14)
Append_Child
  in Ada.Containers.Multiway_Trees A.18.10(52/5)
Append_Vector
  in Ada.Containers.Vectors A.18.2(46/5)
applicable index constraint 4.3.3(10/5)
application areas 1.1.2(7)
applies
  aspect 13.1.1(23/3), 13.1.1(27/3), 13.1.1(29/3), 13.1.1(30/3)
apply
  to a callable construct by a return statement 6.5(4/2)
  to a loop_statement by an exit_statement 5.7(4)
  to a program unit by a program unit pragma J.15(2/5)
arbitrary order 1.1.4(18)
  allowed 2.8(12), 3.3.1(20/2), 3.5(9), 3.6(22/2), 3.11(10/1), 3.11(11/3), 3.11(13), 4.1.1(7), 4.1.2(7), 4.3(5), 4.3.1(19/5), 4.3.2(7), 4.3.3(20.3/5), 4.3.3(22), 4.3.3(23), 4.3.5(43/5), 4.3.5(44/5), 4.3.5(45/5), 4.3.5(46/5), 4.5.2(27/4), 4.5.10(24/5), 4.8(10/2), 5.2(7), 6.1.1(34/5), 6.1.1(35/3), 6.4(10/2), 6.4.1(17), 7.6(12), 7.6(16/3), 7.6.1(9/3), 7.6.1(11.1/3), 7.6.1(20.2/3), 9.7.1(15), 9.8(4), 12.3(20), 13.11.5(7/3)
Arccos
  in Ada.Numerics.Generic_Complex_Elementary_Functions G.1.2(5)
  in Ada.Numerics.Generic_Elementary_Functions A.5.1(6)
Arccosh
  in Ada.Numerics.Generic_Complex_Elementary_Functions G.1.2(7)
  in Ada.Numerics.Generic_Elementary_Functions A.5.1(7)
Arccot
  in Ada.Numerics.Generic_Complex_Elementary_Functions G.1.2(5)
  in Ada.Numerics.Generic_Elementary_Functions A.5.1(6)
Arccoth
  in Ada.Numerics.Generic_Complex_Elementary_Functions G.1.2(7)
  in Ada.Numerics.Generic_Elementary_Functions A.5.1(7)
Arcsin
  in Ada.Numerics.Generic_Complex_Elementary_Functions G.1.2(5)
  in Ada.Numerics.Generic_Elementary_Functions A.5.1(6)
Arcsinh
  in Ada.Numerics.Generic_Complex_Elementary_Functions G.1.2(7)
  in Ada.Numerics.Generic_Elementary_Functions A.5.1(7)
Arctan
  in Ada.Numerics.Generic_Complex_Elementary_Functions G.1.2(5)
  in Ada.Numerics.Generic_Elementary_Functions A.5.1(6)

Arctanh
  in Ada.Numerics.Generic_Complex_-Elementary_Functions G.1.2(7)
  in Ada.Numerics.Generic_Elementary_-Functions A.5.1(7)
ARG 1.3(1.f/5)
Argument
  in Ada.Command_Line A.15(5)
  in Ada.Numerics.Generic_Complex_-Arrays G.3.2(10/2), G.3.2(31/2)
  in Ada.Numerics.Generic_Complex_-Types G.1.1(10)
argument of a pragma 2.8(9)
Argument_Count
  in Ada.Command_Line A.15(4)
Argument_Error
  in Ada.Numerics A.5(3/5)
Arithmetic
  child of Ada.Calendar 9.6.1(8/5)
array 3.6(1)
array component expression 4.3.3(6)
array component iterator 5.5.2(3/3)
array for a loop 5.5.2(11/5)
array indexing
  See indexed_component 4.1.1(1)
array slice 4.1.2(1)
array type 1.3.1(9/5), 3.6(1)
array_aggregate 4.3.3(2/5)
  used 4.3(2/5), 13.4(3), P.1
array_component_association 4.3.3(5/5)
  used 4.3.3(4.1/5), P.1
array_component_association_list 4.3.3(4.1/5)
  used 4.3.3(4/5), 4.3.4(4/5), P.1
array_delta_aggregate 4.3.4(4/5)
  used 4.3.4(2/5), P.1
array_type_definition 3.6(2)
  used 3.2.1(4/2), 3.3.1(2/3), 12.5.3(2), P.1
ASCII
  package physically nested within the declaration of Standard A.1(36.3/2)
  in Standard A.1(36.3/2)
aspect 1.3.1(10/5), 13.1(0.1/3), K.1(1/3)
  additive 13.1.1(18.3/5)
  assertion 11.4.2(9/5)
  class-wide 13.1.1(28/4)
  implicitly composed 13.1.1(18.3/5)
  interfacing B.1(0.1/3)
  library unit 13.1.1(32/5)
  nonoverridable 13.1.1(18.3/5)
  predicate 3.2.4(1/5)
  user-defined literal 4.2.1(2/5)
aspect of representation 13.1(8/5)
aspect_clause 13.1(2/1)
  used 3.8(5/1), 3.11(4/1), 9.1(5/1), 9.4(5/1), 9.4(8/4), P.1
aspect_definition 13.1.1(4/5)
  used 13.1.1(2/3), P.1
aspect_mark 13.1.1(3/3)
  used 2.8(3/3), 11.4.2(6.1/3), 13.1.1(2/3), L(2.4/3), P.1
aspect_specification 13.1.1(2/3)
  used 3.2.1(3/3), 3.2.2(2/3), 3.3.1(2/3), 3.7(5/5), 3.8(6/3), 3.9.3(1.1/3), 4.5.10(3/5), 5.5(3/5), 5.6.1(2/5), 6.1(2/3), 6.1(15/5), 6.3(2/3), 6.5(2.1/5), 6.7(2/3), 6.8(2/4), 7.1(3/3), 7.2(2/3), 7.3(2/3), 7.3(3/3), 8.5.1(2/5), 8.5.2(2/3), 8.5.3(2/3), 8.5.4(2/3), 8.5.5(2/3), 9.1(2/3), 9.1(3/5), 9.1(6/3), 9.4(2/3), 9.4(3/3), 9.4(7/3), 9.5.2(2/3), 9.5.2(5/5), 9.5.2(8/5), 10.1.3(3/3), 10.1.3(4/3), 10.1.3(5/3), 10.1.3(6/3), 11.1(2/3), 12.1(3/3), 12.3(2/3), 12.4(2/3), 12.5(2.1/5), 12.6(2.1/3), 12.6(2.2/3), 12.7(2/3), P.1
aspects
  Address 13.3(12)
  Aggregate 4.3.5(3/5)
  Alignment (subtype) 13.3(26.4/2)
  All_Calls_Remote E.2.3(16/5)
  Asynchronous E.4.1(8.1/3)
  Atomic C.6(6.2/3)
  Atomic_Components C.6(6.7/3)
  Attach_Handler C.3.1(6.3/3)
  Bit_Order 13.5.3(4)
  Coding 13.4(7)
  Component_Size 13.3(70)
  Constant_Indexing 4.1.6(2/5)
  Convention B.1(2/3)
  CPU D.16(8/3)
  Default_Component_Value 3.6(22.2/3)
  Default_Initial_Condition 7.3.3(2/5)
  Default_Iterator 5.5.1(8/5)
  Default_Storage_Pool 13.11.3(5/4)
  Default_Value 3.5(56.3/3)
  Discard_Names C.5(1.3/4)
  Dispatching H.7.1(8/5)
  Dispatching_Domain D.16.1(18/3)
  Dynamic_Predicate 3.2.4(1/5)
  Elaborate_Body 10.2.1(26.1/5)
  Exclusive_Functions 9.5.1(2.2/4)
  Export B.1(1/3)
  External_Name B.1(1/3)
  External_Tag 13.3(75/3), K.2(65)
  Full_Access_Only C.6(6.5/5)
  Global 6.1.2(11/5)
  Global'Class 6.1.2(15/5)
  Implicit_Dereference 4.1.5(2/3)
  Import B.1(1/3)
  Independent C.6(6.3/3)
  Independent_Components C.6(6.10/3)
  Inline 6.3.2(5.1/3)
  Input 13.13.2(38/5)
  Input'Class 13.13.2(38/5)
  Integer_Literal 4.2.1(3/5)
  Interrupt_Handler C.3.1(6.2/3)
  Interrupt_Priority D.1(6.3/3)
  Iterator_Element 5.5.1(9/5)
  Iterator_View 5.5.1(9.1/5)
  Layout 13.5(1)
  Link_Name B.1(1/3)
  Machine_Radix F.1(1/5)
  Max_Entry_Queue_Length D.4(18/5)
  No_Controlled_Parts H.4.1(2/5)
  No_Return 6.5.1(3.2/3)
  Nonblocking 9.5(19/5)
  Output 13.13.2(38/5)
  Output'Class 13.13.2(38/5)
  Pack 13.2(5.1/3)
  Parallel_Calls 9.10.1(24/5)
  Post 6.1.1(4/5)
  Post'Class 6.1.1(5/5)
  Pre 6.1.1(2/5)
  Pre'Class 6.1.1(3/5)
  Predicate_Failure 3.2.4(14.2/4)
  Preelaborate 10.2.1(11/5)
  Priority D.1(6.2/3)
  Put_Image 4.10(4/5)
  Read 13.13.2(38/5)
  Read'Class 13.13.2(38/5)
  Real_Literal 4.2.1(4/5)
  Record layout 13.5(1)
  Relative_Deadline D.2.6(9.2/3)
  Remote_Call_Interface E.2.3(7/5)
  Remote_Types E.2.2(4/5)
  Shared_Passive E.2.1(4/5)
  Size (object) 13.3(41)
  Size (subtype) 13.3(48)
  Small 3.5.10(2/5)
  Stable_Properties 7.3.4(5/5)
  Stable_Properties'Class 7.3.4(6/5)
  Static 6.8(3.6/5)
  Static_Predicate 3.2.4(1/5)
  Storage_Pool 13.11(15/5)
  Storage_Size (access) 13.11(15/5)
  Storage_Size (task) 13.3(65.2/3)
  Stream_Size 13.13.2(1.5/2)
  String_Literal 4.2.1(5/5)
  Synchronization 9.5(12/3)
  Type_Invariant 7.3.2(2/3)
  Type_Invariant'Class 7.3.2(3/5)
  Unchecked_Union B.3.3(3.2/3)
  Use_Formal H.7.1(3/5)
  Variable_Indexing 4.1.6(3/5)
  Volatile C.6(6.4/3)
  Volatile_Components C.6(6.8/3)
  Write 13.13.2(38/5)
  Write'Class 13.13.2(38/5)
  Yield D.2.1(1.7/5)
assembly language C.1(4/3)
Assert
  in Ada.Assertions 11.4.2(14/2)
Assert pragma 11.4.2(3/2), L(2.2/2)
assertion 1.3.4(1/5)
assertion aspect 11.4.2(9/5)
assertion expressions 11.4.2(1.1/5)
assertion policy
  Assert pragma 11.4.2(18/3)
Assertion_Error
  raised by failure of assertion 11.4.2(18/3)
  raised by failure of runtime check 3.2.4(31.1/4), 4.6(57/5), 6.1.1(32/3), 6.1.1(33/3), 6.1.1(35/3), 6.5(22/5), 7.3.2(22/3), 7.3.3(9/5), 11.5(23.3/5)
  in Ada.Assertions 11.4.2(13/2)
Assertion_Policy pragma 11.4.2(6.1/3), 11.4.2(6/2), L(2.3/2), L(2.4/3)
assertions 11.4.2(1.1/5)

child of Ada 11.4.2(12/5)
Assign
in Ada.Containers.Doubly_Linked_-Lists A.18.3(17.5/5), A.18.3(51.8/5)
in Ada.Containers.Hashed_Maps A.18.5(17.7/5), A.18.5(37.9/5)
in Ada.Containers.Hashed_Sets A.18.8(17.4/5), A.18.8(59.8/5)
in Ada.Containers.Indefinite_Holders A.18.18(20/5)
in Ada.Containers.Multiway_Trees A.18.10(32/5), A.18.10(70.8/5)
in Ada.Containers.Ordered_Maps A.18.6(16.7/5), A.18.6(51.10/5)
in Ada.Containers.Ordered_Sets A.18.9(16.4/5), A.18.9(74.8/5)
in Ada.Containers.Vectors A.18.2(34.7/5), A.18.2(79.8/5)
See assignment operation 5.2(3)
Assign_Task
in System.Multiprocessors.Dispatching_Domains D.16.1(11/3)
assigning back of parameters 6.4.1(17)
assignment
user-defined 7.6(1)
assignment operation 5.2(3), 5.2(12/5), 7.6(13)
during elaboration of an object_declaration 3.3.1(18/2)
during evaluation of a generic_association for a formal object of mode in 12.4(11)
during evaluation of a parameter_association 6.4.1(11)
during evaluation of an aggregate 4.3(5)
during evaluation of an initialized allocator 4.8(7/2)
during evaluation of an uninitialized allocator 4.8(9/2)
during evaluation of concatenation 4.5.3(10)
during execution of a for loop 5.5(9/5)
during execution of an assignment_statement 5.2(12/5)
during parameter copy back 6.4.1(17)
list of uses 7.6.1(24.d)
assignment_statement 5.2(2)
used 5.1(4/2), P.1
associated components
of a record_component_association 4.3.1(10)
associated declaration
of an aspect specification 13.1.1(1/3)
associated discriminants
of a named discriminant_association 3.7.1(5)
of a positional discriminant_association 3.7.1(5)
associated entity
of an aspect specification 13.1.1(5/5)
associated object
of a value of a by-reference type 6.2(10/5)
of a value of a limited type 6.2(10.f)
asterisk 2.1(15/5)
in Ada.Characters.Latin_1 A.3.3(8)
asynchronous
remote procedure call E.4.1(9/3)
Asynchronous aspect E.4.1(8.1/3)
Asynchronous pragma J.15.13(2/3), L(3.1/3)
asynchronous remote procedure call E.4(1/5)
asynchronous_select 9.7.4(2)
used 9.7(2), P.1
Asynchronous_Task_Control
child of Ada D.11(3/5)
at sign 2.1(15/5)
at-most-once execution E.4(11)
at_clause J.7(1)
used 13.1(2/1), P.1
atomic C.6(7/3)
Atomic aspect C.6(6.2/3)
Atomic pragma J.15.8(2/3), L(4.1/3)
Atomic_Add
in System.Atomic_Operations.Integer_Arithmetic C.6.4(4/5)
in System.Atomic_Operations.Modular_Arithmetic C.6.5(4/5)
Atomic_Clear
in System.Atomic_Operations.Test_And_Set C.6.3(6/5)
Atomic_Compare_And_Exchange
in System.Atomic_Operations.Exchange C.6.2(7/5)
Atomic_Components aspect C.6(6.7/3)
Atomic_Components pragma J.15.8(5/3), L(5.1/3)
Atomic_Exchange
in System.Atomic_Operations.Exchange C.6.2(6/5)
Atomic_Fetch_And_Add
in System.Atomic_Operations.Integer_Arithmetic C.6.4(6/5)
in System.Atomic_Operations.Modular_Arithmetic C.6.5(6/5)
Atomic_Fetch_And_Subtract
in System.Atomic_Operations.Integer_Arithmetic C.6.4(7/5)
in System.Atomic_Operations.Modular_Arithmetic C.6.5(7/5)
Atomic_Operations
child of System C.6.1(3/5)
Atomic_Subtract
in System.Atomic_Operations.Integer_Arithmetic C.6.4(5/5)
in System.Atomic_Operations.Modular_Arithmetic C.6.5(5/5)
Atomic_Test_And_Set
in System.Atomic_Operations.Test_And_Set C.6.3(5/5)
Attach_Handler
in Ada.Interrupts C.3.2(7)
Attach_Handler aspect C.3.1(6.3/3)
Attach_Handler pragma J.15.7(4/3), L(6.1/3)
attaching
to an interrupt C.3(2)
attribute 1.3.1(12/5), 4.1.4(1), K.2(1/3)
representation 13.3(1/1)
specifiable 13.3(5/3)
specifying 13.3(1/1)
stream-oriented 13.13.2(1/3)
attribute_definition_clause 13.3(2)
used 13.1(2/1), P.1
attribute_designator 4.1.4(3/2)
used 4.1.4(2/5), 13.1(3), 13.3(2), P.1
Attribute_Handle
in Ada.Task_Attributes C.7.2(3)
attribute_reference 4.1.4(2/5)
used 4.1(2/5), P.1
attributes
Access 3.10.2(24/1), 3.10.2(32/5)
Address 13.3(11), J.7.1(5)
Adjacent A.5.3(48)
Aft 3.5.10(5)
Alignment 13.3(23/2), 13.3(26.2/2)
Base 3.5(15)
Bit_Order 13.5.3(4)
Body_Version E.3(4)
Callable 9.9(2)
Caller C.7.1(14/3)
Ceiling A.5.3(33)
Class 3.9(14), 7.3.1(9), J.11(2/2)
Component_Size 13.3(69)
Compose A.5.3(24)
Constrained 3.7.2(3/5), J.4(2)
Copy_Sign A.5.3(51)
Count 9.9(5)
Definite 12.5.1(23/3)
Delta 3.5.10(3)
Denorm A.5.3(9)
Digits 3.5.8(2/1), 3.5.10(7)
Enum_Rep 13.4(10.2/5)
Enum_Val 13.4(10.5/5)
Exponent A.5.3(18)
External_Tag 13.3(75/3)
First 3.5(12), 3.6.2(3)
First(N) 3.6.2(4)
First_Bit 13.5.2(3/2)
First_Valid 3.5.5(7.2/4)
Floor A.5.3(30)
Fore 3.5.10(4)
Fraction A.5.3(21)

Has_Same_Storage 13.3(73.2/4)
Identity 11.4.1(9), C.7.1(12)
Image 4.10(34/5), 4.10(40/5)
Index 6.1.1(30.2/5)
Input 13.13.2(22), 13.13.2(32)
Last 3.5(13), 3.6.2(5)
Last(N) 3.6.2(6)
Last_Bit 13.5.2(4/2)
Last_Valid 3.5.5(7.3/4)
Leading_Part A.5.3(54)
Length 3.6.2(9)
Length(N) 3.6.2(10)
Machine A.5.3(60)
Machine_Emax A.5.3(8)
Machine_Emin A.5.3(7)
Machine_Mantissa A.5.3(6)
Machine_Overflows A.5.3(12), A.5.4(4)
Machine_Radix A.5.3(2), A.5.4(2)
Machine_Rounding A.5.3(41.1/2)
Machine_Rounds A.5.3(11), A.5.4(3)
Max 3.5(19)
Max_Alignment_For_Allocation 13.11.1(4/5)
Max_Size_In_Storage_Elements 13.11.1(3/5)
Min 3.5(16)
Mod 3.5.4(16.1/2)
Model A.5.3(68), G.2.2(7)
Model_Emin A.5.3(65), G.2.2(4)
Model_Epsilon A.5.3(66)
Model_Mantissa A.5.3(64), G.2.2(3/2)
Model_Small A.5.3(67)
Modulus 3.5.4(17)
Object_Size 13.3(58.2/5)
Old 6.1.1(26/5)
Output 13.13.2(19), 13.13.2(29)
Overlaps_Storage 13.3(73.6/3)
Parallel_Reduce(Reducer, Initial_Value) 4.5.10(33/5)
Partition_Id E.1(9)
Pos 3.5.5(2)
Position 13.5.2(2/2)
Pred 3.5(25)
Preelaborable_Initialization 10.2.1(11.7/5)
Priority D.5.2(3/2)
Put_Image 4.10(3/5)
Range 3.5(14), 3.6.2(7)
Range(N) 3.6.2(8)
Read 13.13.2(6), 13.13.2(14)
Reduce(Reducer, Initial_Value) 4.5.10(23/5), 4.5.10(32/5)
Relative_Deadline D.5.2(3.1/5)
Remainder A.5.3(45)
Result 6.1.1(29/5)
Round 3.5.10(12)
Rounding A.5.3(36)
Safe_First A.5.3(71), G.2.2(5)
Safe_Last A.5.3(72), G.2.2(6)
Scale 3.5.10(11)
Scaling A.5.3(27)
Signed_Zeros A.5.3(13)
Size 13.3(40), 13.3(45)
Small 3.5.10(2/5)
Storage_Pool 13.11(13)
Storage_Size 13.3(60/3), 13.11(14), J.9(2)
Stream_Size 13.13.2(1.2/3)
Succ 3.5(22)
Tag 3.9(16), 3.9(18)
Terminated 9.9(3)
Truncation A.5.3(42)
Unbiased_Rounding A.5.3(39)
Unchecked_Access 13.10(3), H.4(18)
Val 3.5.5(5)
Valid 13.9.2(3/4), H(6/5)
Value 3.5(52)
Version E.3(3)
Wide_Image 4.10(31/5), 4.10(39/5)
Wide_Value 3.5(40)
Wide_Wide_Image 4.10(30/5), 4.10(38/5)
Wide_Wide_Value 3.5(39.1/2)
Wide_Wide_Width 3.5(37.1/5)
Wide_Width 3.5(38/5)
Width 3.5(39/5)
Write 13.13.2(3), 13.13.2(11)
available
stream attribute 13.13.2(39/2)
avoid overspecifying environmental issues 10(3.a)

B

Backus-Naur Form (BNF)
complete listing P.1(1/5)
cross reference P.2(1/3)
notation 1.1.4(3)
under Syntax heading 1.1.2(25)
Barrier_Limit *subtype of* Positive
in Ada.Synchronous_Barriers D.10.1(4/3)
base 2.4.2(3), 2.4.2(6)
absolute deadline D.2.6(14/5)
used 2.4.2(2), P.1
base 16 literal 2.4.2(1)
base 2 literal 2.4.2(1)
base 8 literal 2.4.2(1)
Base attribute 3.5(15)
base decimal precision
of a floating point type 3.5.7(9)
of a floating point type 3.5.7(10/5)
base priority D.1(15/5)
base range
of a decimal fixed point type 3.5.9(16)
of a fixed point type 3.5.9(12)
of a floating point type 3.5.7(8), 3.5.7(10/5)
of a modular type 3.5.4(10)
of a scalar type 3.5(6)
of a signed integer type 3.5.4(9)
of an enumeration type 3.5(6.b)
of an ordinary fixed point type 3.5.9(13)
base subtype
of a type 3.5(15)
Base_Name
in Ada.Directories A.16(19/5)
based_literal 2.4.2(2)
used 2.4(2), P.1
based_numeral 2.4.2(4)
used 2.4.2(2), P.1
basic letter
a category of Character A.3.2(27)
basic_declaration 3.1(3/3)
used 3.11(4/1), P.1
basic_declarative_item 3.11(4/1)
used 3.11(3), 7.1(3/3), P.1
basic_global_mode 6.1.2(5/5)
used 6.1.2(4/5), H.7(3/5), P.1
Basic_Map
in Ada.Strings.Maps.Constants A.4.6(5)
Basic_Set
in Ada.Strings.Maps.Constants A.4.6(4)
Beaujolais effect 8.4(1.b)
[*partial*] 3.6(18.b), 8.6(22.a), 8.6(34.a), 8.6(34.k)
become nonlimited 7.3.1(5/1), 7.5(16)
BEL
in Ada.Characters.Latin_1 A.3.3(5)
belong
to a range 3.5(4)
to a subtype 3.2(8/5)
belongs
subpool to a pool 13.11.4(20/4)
bibliography 1.2.1(1/5)
big endian 13.5.3(2)
big-O notation A.18(3.b/2)
Big_Integer
in Ada.Numerics.Big_Numbers.Big_Integers A.5.6(3/5)
Big_Integers
child of Ada.Numerics.Big_Numbers A.5.5(3/5), A.5.6(2/5)
Big_Natural *subtype of* Big_Integer
in Ada.Numerics.Big_Numbers.Big_Integers A.5.6(9/5)
Big_Positive *subtype of* Big_Integer
in Ada.Numerics.Big_Numbers.Big_Integers A.5.6(8/5)
Big_Real
in Ada.Numerics.Big_Numbers.Big_Reals A.5.7(3/5)
Big_Reals
child of Ada.Numerics.Big_Numbers A.5.7(2/5)
binary
literal 2.4.2(1)
in Interfaces.COBOL B.4(10)
binary adding operator 4.5.3(1)
binary literal 2.4.2(1)
binary operator 4.5(9)
binary_adding_operator 4.5(4)
used 4.4(4), P.1

Binary_Format
  in Interfaces.COBOL B.4(24)
bit field
  See record_representation_clause 13.5.1(1)
bit ordering 13.5.3(2)
bit string
  See logical operators on boolean arrays 4.5.1(2)
Bit_Order
  in System 13.7(15/2)
Bit_Order aspect 13.5.3(4)
Bit_Order attribute 13.5.3(4)
Bit_Order clause 13.3(7/2), 13.5.3(4)
blank
  in text input for enumeration and numeric types A.10.6(5/2)
Blank_When_Zero
  in Ada.Text_IO.Editing F.3.3(7)
block_statement 5.6(2)
  used 5.1(5/5), P.1
blocked
  [*partial*] D.2.1(11)
  a task state 9(10/5)
  during an entry call 9.5.3(19)
  execution of a selective_accept 9.7.1(16)
  on a delay_statement 9.6(21)
  on an accept_statement 9.5.2(24/5)
  waiting for activations to complete 9.2(5)
  waiting for dependents to terminate 9.3(5)
blocked interrupt C.3(2)
blocking
  allows 9.5(21/5)
  non- 9.5(21/5)
blocking, potentially 9.5(34/5)
  delay_statement 9.6(34), D.9(5)
  remote subprogram call E.4(17/5)
BMP 3.5.2(2/5), 3.5.2(3/5)
BNF (Backus-Naur Form)
  complete listing P.1(1/5)
  cross reference P.2(1/3)
  notation 1.1.4(3)
  under Syntax heading 1.1.2(25)
body 3.11(5), 3.11.1(1/3)
  used 3.11(3), P.1
body_stub 10.1.3(2)
  used 3.11(5), P.1
Body_Version attribute E.3(4)
BOM_16
  in Ada.Strings.UTF_Encoding A.4.11(12/3)
BOM_16BE
  in Ada.Strings.UTF_Encoding A.4.11(10/3)
BOM_16LE
  in Ada.Strings.UTF_Encoding A.4.11(11/3)
BOM_8
  in Ada.Strings.UTF_Encoding A.4.11(9/3)
Boolean 3.5.3(1)
  in Standard A.1(5)
boolean type 3.5.3(1)
boundary entity 7.3.2(8.3/5)
Bounded
  child of Ada.Streams.Storage 13.13.1(26/5)
  child of Ada.Strings A.4.4(3/5)
  child of Ada.Strings.Text_Buffers A.4.12(26/5)
bounded error 1.1.2(31), 1.1.5(8/5)
  cause 4.2.1(17/5), 4.5.10(34/5), 4.8(11.1/2), 5.1(18/5), 5.5.3(28/5), 6.2(12/3), 7.6.1(14/1), 9.4(20.1/2), 9.5.1(8/5), 9.5.1(18/5), 9.8(20/5), 10.2(26), 13.9.1(9), 13.11.2(11), 13.11.2(15.2/4), A.5.2(40.1/5), A.17(25/2), A.18.2(238/3), A.18.2(239.1/3), A.18.2(239.2/3), A.18.2(239/2), A.18.2(243/2), A.18.3(152.1/3), A.18.3(152.2/3), A.18.3(152/2), A.18.4(75.7/3), A.18.4(75.8/3), A.18.7(96.19/3), A.18.7(96.20/3), A.18.10(220/3), A.18.10(221/3), A.18.18(68/3), A.18.18(69/3), A.18.19(10/3), A.18.20(14/3), A.18.21(15/3), A.18.22(12/3), A.18.23(15/3), A.18.24(12/3), A.18.25(14/3), A.18.32(11/5), C.7.1(17/3), C.7.2(13.2/1), D.3(13.6/5), E.1(10/2), E.3(6), J.7.1(11)
Bounded_Doubly_Linked_Lists
  child of Ada.Containers A.18.20(1/3)
Bounded_Hashed_Maps
  child of Ada.Containers A.18.21(1/3)
Bounded_Hashed_Sets
  child of Ada.Containers A.18.23(1/3)
Bounded_Indefinite_Holders
  child of Ada.Containers A.18.32(1/5)
Bounded_IO
  child of Ada.Text_IO A.10.11(3/5)
Bounded_Multiway_Trees
  child of Ada.Containers A.18.25(1/3)
Bounded_Ordered_Maps
  child of Ada.Containers A.18.22(1/3)
Bounded_Ordered_Sets
  child of Ada.Containers A.18.24(1/3)
Bounded_Priority_Queues
  child of Ada.Containers A.18.31(2/5)
Bounded_Slice
  in Ada.Strings.Bounded A.4.4(28.1/2), A.4.4(28.2/2)
Bounded_String
  in Ada.Strings.Bounded A.4.4(6)
Bounded_Synchronized_Queues
  child of Ada.Containers A.18.29(2/5)
Bounded_Vectors
  child of Ada.Containers A.18.19(1/3)
bounds
  of a discrete_range 3.6.1(6)
  of an array 3.6(13)
  of the index range of an array_aggregate 4.3.3(24)
box
  compound delimiter 3.6(15/5)
BPH
  in Ada.Characters.Latin_1 A.3.3(17)
broadcast signal
  See protected object 9.4(1)
  See requeue 9.5.4(1)
Broken_Bar
  in Ada.Characters.Latin_1 A.3.3(21/3)
BS
  in Ada.Characters.Latin_1 A.3.3(5)
budget D.14.2(14/2)
Budget_Has_Expired
  in Ada.Execution_Time.Group_Budgets D.14.2(9/2)
Budget_Remaining
  in Ada.Execution_Time.Group_Budgets D.14.2(9/2)
Buffer_Size
  in Ada.Storage_IO A.9(4)
Buffer_Type
  in Ada.Strings.Text_Buffers.Bounded A.4.12(27/5)
  in Ada.Strings.Text_Buffers.Unbounded A.4.12(19/5)
Buffer_Type *subtype of* Storage_Array
  in Ada.Storage_IO A.9(4)
build-in-place
  See built in place
built in place 7.6(17.1/5)
by copy parameter passing 6.2(2)
by reference parameter passing 6.2(2)
by-copy type 6.2(3/3)
by-reference primitive 13.1(10/5)
by-reference type 6.2(4)
  atomic or volatile C.6(18)
Byte
  in Interfaces.COBOL B.4(29/3)
  See storage element 13.3(8)
byte sex
  See ordering of storage elements in a word 13.5.3(5)
Byte_Array
  in Interfaces.COBOL B.4(29/3)

C

C
  child of Interfaces B.3(4/5)
C interface B.3(1/4)
C standard 1.2.1(8/5)
C++ standard 1.2.1(9/5)
C_bool
  in Interfaces.C B.3(13.1/5)
C_float
  in Interfaces.C B.3(15)
C_Variadic B.3(60.16/4)
Calendar
  child of Ada 9.6(10/5)
Calendar_Assertion_Check 11.5(23.4/5)
call 6(2/3)

master of 3.10.2(10.1/3)
call on a dispatching operation 3.9.2(2/3)
callable 9.9(2)
Callable attribute 9.9(2)
callable construct 6(2/3)
callable entity 6(2/3)
called partition E.4(1/5)
Caller attribute C.7.1(14/3)
calling convention 6.3.1(2/1), B.1(11/3)
Ada 6.3.1(3/3)
associated with a designated profile 3.10(11)
entry 6.3.1(13/4)
Intrinsic 6.3.1(4)
protected 6.3.1(12/4)
calling partition E.4(1/5)
calling stub E.4(10)
CAN
in Ada.Characters.Latin_1 A.3.3(6)
can cause 13.1(28/5)
cancel
a logical thread 5.1(16/5)
a parallel construct 5.1(16/5)
Cancel_Handler
in Ada.Execution_Time.Group_Budgets D.14.2(10/2)
in Ada.Execution_Time.Timers D.14.1(7/2)
in Ada.Real_Time.Timing_Events D.15(5/2)
cancellation
of a delay_statement 9.6(22/3)
of an entry call 9.5.3(20)
cancellation of a remote subprogram call E.4(13)
canonical form A.5.3(3)
canonical order of array components 5.5.2(11/5)
canonical semantics 11.6(2/3)
canonical-form representation A.5.3(10)
capacity
of a hashed map A.18.5(41/2)
of a hashed set A.18.8(63/2)
of a queue A.18.27(10/3)
of a vector A.18.2(2/2)
in Ada.Containers.Hashed_Maps A.18.5(8/5)
in Ada.Containers.Hashed_Sets A.18.8(10/5)
in Ada.Containers.Vectors A.18.2(19/5)
Capacity_Error
in Ada.Containers A.18.1(5.1/3)
case expression 4.5.7(1/3)
case insensitive 2.3(5/5)
case_expression 4.5.7(5/3)
used 4.5.7(2/3), P.1
case_expression_alternative 4.5.7(6/3)
used 4.5.7(5/3), P.1
case_statement 5.4(2/3)
used 5.1(5/5), P.1
case_statement_alternative 5.4(3)
used 5.4(2/3), P.1
cast
See type conversion 4.6(1/3)
See unchecked type conversion 13.9(1)
catch (an exception)
See handle 11(1/3)
categorization aspect E.2(2/3)
categorization pragma J.15.15(8/5)
Pure J.15.15(9/5)
Remote_Call_Interface J.15.15(9/5)
Remote_Types J.15.15(9/5)
Shared_Passive J.15.15(9/5)
categorized library unit E.2(2/3)
category
of types 3.2(2/5), 3.4(1.1/2)
category determined for a formal type 12.5(6/3)
category of types 1.3.1(13/5)
catenation operator
See concatenation operator 4.5(1)
See concatenation operator 4.5.3(3)
Cause_Of_Termination
in Ada.Task_Termination C.7.3(3/2)
CCH
in Ada.Characters.Latin_1 A.3.3(18)
cease to exist
object 7.6.1(11/3), 13.11.2(10/4)
type 7.6.1(11/3)
Cedilla
in Ada.Characters.Latin_1 A.3.3(22)
Ceiling
in Ada.Containers.Ordered_Maps A.18.6(41/5)
in Ada.Containers.Ordered_Sets A.18.9(51/2), A.18.9(71/5)
Ceiling attribute A.5.3(33)
ceiling priority
of a protected object D.3(8/3)
Ceiling_Check
[*partial*] C.3.1(11/3), D.3(13), D.3(13.5/5)
Ceiling_Locking locking policy D.3(7)
Cent_Sign
in Ada.Characters.Latin_1 A.3.3(21/3)
change of representation 13.6(1/5)
char
in Interfaces.C B.3(19)
char16_array
in Interfaces.C B.3(39.5/3)
char16_nul
in Interfaces.C B.3(39.3/2)
char16_t
in Interfaces.C B.3(39.2/2)
char32_array
in Interfaces.C B.3(39.14/3)
char32_nul
in Interfaces.C B.3(39.12/2)
char32_t
in Interfaces.C B.3(39.11/2)
char_array
in Interfaces.C B.3(23/3)
char_array_access
in Interfaces.C.Strings B.3.1(4)
CHAR_BIT
in Interfaces.C B.3(6)
Character 3.5.2(2/5)
used 2.7(2), P.1
in Standard A.1(35/3)
character encoding A.4.11(46/3)
character plane 2.1(1/5)
character set 2.1(1/5)
character set standard
16 and 32-bit 1.2(8/5)
7-bit 1.2.1(2/5)
8-bit 1.2.1(7/5)
control functions 1.2.1(5/5)
character type 1.3.1(15/5), 3.5.2(1)
character_literal 2.5(2)
used 3.5.1(4), 4.1(2/5), 4.1.3(3), P.1
Character_Mapping
in Ada.Strings.Maps A.4.2(20/5)
Character_Mapping_Function
in Ada.Strings.Maps A.4.2(25)
Character_Range
in Ada.Strings.Maps A.4.2(6)
Character_Ranges
in Ada.Strings.Maps A.4.2(7)
Character_Sequence *subtype of* String
in Ada.Strings.Maps A.4.2(16)
Character_Set
in Ada.Strings.Maps A.4.2(4/5)
in Ada.Strings.Wide_Maps A.4.7(46/2)
in Ada.Strings.Wide_Maps.Wide_-Constants A.4.8(48/2)
in Interfaces.Fortran B.5(11)
Character_Set_Version
in Ada.Wide_Characters.Handling A.3.5(4/3)
characteristics
[*partial*] 3.4(7/3)
Characters
child of Ada A.3.1(2/5)
Characters_Assertion_Check 11.5(23.5/5)
chars_ptr
in Interfaces.C.Strings B.3.1(5/5)
chars_ptr_array
in Interfaces.C.Strings B.3.1(6/2)
check 1.3.5(1/5)
language-defined 11.5(2/3), 11.6(1/3)
check, language-defined
Access_Check 4.1(13), 4.1.5(8/3), 4.6(51/5), 4.8(10.4/3)
Accessibility_Check 3.10.2(29), 4.6(39.1/2), 4.6(48/3), 4.8(10.1/3), 6.5(8/5), 6.5(21/3), 9.5.4(7.2/5), 13.11.4(25/3), 13.11.4(26/3), E.4(18/1)
Allocation_Check 4.8(10.2/2), 4.8(10.3/2), 4.8(10.4/3), 13.11.4(30/3)
Ceiling_Check C.3.1(11/3), D.3(13), D.3(13.5/5)
controlled by assertion policy 3.2.4(31/5), 4.6(51/5), 6.1.1(32/3), 6.1.1(33/3), 6.1.1(35/3), 7.3.2(22/3), 7.3.3(9/5)
Discriminant_Check 4.1.3(15), 4.3(6), 4.3.2(8/3), 4.3.4(16/5), 4.6(43), 4.6(45), 4.6(51/5), 4.6(52), 4.7(4/5), 4.8(10/2), 6.5(5.12/5)

Division_Check 3.5.4(20), 4.5.5(22), A.5.3(47), G.1.1(40), K.2(202)
Elaboration_Check 3.11(9)
Index_Check 4.1.1(7), 4.1.2(7), 4.3.3(20.4/5), 4.3.3(29/3), 4.3.3(30), 4.3.4(20/5), 4.5.3(8), 4.6(51/5), 4.7(4/5), 4.8(10/2)
Length_Check 4.5.1(8), 4.6(37), 4.6(52)
Overflow_Check 3.5.4(20), 4.4(11), G.2.1(11), G.2.2(7), G.2.3(25), G.2.4(2), G.2.6(3/5)
Partition_Check E.4(19)
Program_Error_Check 3.2.4(29.1/4), 5.5(8.1/5), 5.6.1(4/5), 12.5.1(23.3/2), 13.3(75.1/3), B.3.3(22/2), H.4(23.11/5)
Range_Check 3.2.2(11), 3.5(24), 3.5(27), 3.5(39.12/3), 3.5(39.4/3), 3.5(39.5/3), 3.5(43/5), 3.5(55/5), 3.5.5(7), 3.5.9(19/4), 4.2(11/5), 4.3.3(28), 4.5.1(8), 4.5.6(6), 4.5.6(13), 4.6(28), 4.6(38), 4.6(46), 4.6(51/5), 4.7(4/5), 13.13.2(35/3), A.5.3(26), A.5.3(29), A.5.3(50), A.5.3(53), A.5.3(59), A.5.3(62), K.2(11), K.2(114), K.2(122), K.2(184), K.2(220), K.2(241), K.2(41), K.2(47)
Reserved_Check C.3.1(10/3)
Storage_Check 11.1(6/5), 13.3(67), 13.11(17), D.7(17/1), D.7(18/1), D.7(19/1)
Tag_Check 3.9.2(16), 4.6(42), 4.6(52), 5.2(10), 6.5(8.2/5)
Tasking_Check 9.2(5), 9.5.3(21)
checking pragmas 11.5(1/2)
child
of a library unit 10.1.1(1)
Child_Count
in Ada.Containers.Multiway_Trees A.18.10(46.1/5), A.18.10(46/5)
Child_Depth
in Ada.Containers.Multiway_Trees A.18.10(47.1/5), A.18.10(47/5)
choice
of an exception_handler 11.2(5.b/3)
choice parameter 11.2(9)
choice_expression 4.4(2.1/3)
used 3.8.1(5/3), P.1
choice_parameter_specification 11.2(4)
used 11.2(3/5), P.1
choice_relation 4.4(2.2/3)
used 4.4(2.1/3), P.1
chunk 5.5(8.1/5)
[*partial*] 5.6.1(5/5)
chunk parameter 5.5(6.1/5)
Chunk_Count
in Ada.Iterator_Interfaces 5.5.1(4.5/5)
Chunk_Index *subtype of* Positive
in Ada.Iterator_Interfaces 5.5.1(4.2/5)
chunk_specification 5.5(3.1/5)
used 4.5.10(3/5), 5.5(3/5), 5.6.1(2/5), P.1
Circumflex
in Ada.Characters.Latin_1 A.3.3(12)
class
of types 3.2(2/5), 3.4(1.1/2)
See also package 7(1)
See also tag 3.9(3)
Class attribute 3.9(14), 7.3.1(9), J.11(2/2)
class factory 3.9(30/2)
class of types 1.3.1(16/5)
class-wide aspect 13.1.1(28/4)
class-wide postcondition expression 6.1.1(5/5)
class-wide precondition expression 6.1.1(3/5)
class-wide stable property function 7.3.4(6/5)
class-wide type 3.4.1(4), 3.7(26)
class-wide type invariant 7.3.2(3/5)
cleanup
See finalization 7.6.1(1)
clear
execution timer object D.14.1(12/2)
group budget object D.14.2(15/2)
timing event object D.15(9/2)
in Ada.Containers.Doubly_Linked_-Lists A.18.3(13/5)
in Ada.Containers.Hashed_Maps A.18.5(12/5)
in Ada.Containers.Hashed_Sets A.18.8(14/5)
in Ada.Containers.Indefinite_Holders A.18.18(11/5)
in Ada.Containers.Multiway_Trees A.18.10(23/5)
in Ada.Containers.Ordered_Maps A.18.6(11/5)
in Ada.Containers.Ordered_Sets A.18.9(13/5)
in Ada.Containers.Vectors A.18.2(24/5)
in Ada.Environment_Variables A.17(7/2)
in Ada.Streams.Storage 13.13.1(15/5)
in Ada.Streams.Storage.Bounded 13.13.1(31/5)
in Ada.Streams.Storage.Unbounded 13.13.1(23/5)
cleared
termination handler C.7.3(9/2)
clock 9.6(6/3)
in Ada.Calendar 9.6(12)
in Ada.Execution_Time D.14(5/2)
in Ada.Execution_Time.Interrupts D.14.3(3/5)
in Ada.Real_Time D.8(6)
clock jump D.8(32)
clock tick D.8(23)
Clock_For_Interrupts
in Ada.Execution_Time D.14(9.3/3)
Close
in Ada.Direct_IO A.8.4(8)
in Ada.Sequential_IO A.8.1(8)
in Ada.Streams.Stream_IO A.12.1(10)
in Ada.Text_IO A.10.1(11)
close result set G.2.3(5)
closed entry 9.5.3(5)
of a protected object 9.5.3(7/3)
of a task 9.5.3(6/3)
closed under derivation 1.3.1(16/5), 3.2(2.b/5), 3.4(28)
closure
downward 3.10.2(13.b/2), 3.10.2(37/5)
COBOL
child of Interfaces B.4(7/5)
COBOL interface B.4(1/3)
COBOL standard 1.2.1(4/5)
COBOL_Character
in Interfaces.COBOL B.4(13)
COBOL_To_Ada
in Interfaces.COBOL B.4(15)
code point
for characters 3.5.2(2/5), 3.5.2(11.p/3)
code_statement 13.8(2)
used 5.1(4/2), P.1
Coding aspect 13.4(7)
coextension
of an object 3.10.2(14.4/3)
Col
in Ada.Text_IO A.10.1(37/5)
collection
of an access type 7.6.1(11.1/3)
colon 2.1(15/5)
in Ada.Characters.Latin_1 A.3.3(10)
column number A.10(9)
comma 2.1(15/5)
in Ada.Characters.Latin_1 A.3.3(8)
Command_Line
child of Ada A.15(3/5)
Command_Name
in Ada.Command_Line A.15(6)
comment 2.7(2)
comments, instructions for submission 0.2(57.22/5)
Commercial_At
in Ada.Characters.Latin_1 A.3.3(10)
Communication_Error
in System.RPC E.5(5)
comparison operator
See relational operator 4.5.2(1)
compatibility
composite_constraint with an access subtype 3.10(15/2)
constraint with a subtype 3.2.2(12)
delta_constraint with an ordinary fixed point subtype J.3(9/4)
digits_constraint with a decimal fixed point subtype 3.5.9(18/4)
digits_constraint with a floating point subtype J.3(10/4)
discriminant constraint with a subtype 3.7.1(10)
index constraint with a subtype 3.6.1(7)
range with a scalar subtype 3.5(8)
range_constraint with a scalar subtype 3.5(8)
compatible
a type, with a convention B.1(12)
compilation 10.1.1(2)
separate 10.1(1)

compilation unit 1.3.3(2/5), 10.1(2), 10.1.1(9)
compilation units needed
  by a compilation unit 10.2(2/5)
  remote call interface E.2.3(18)
  shared passive library unit E.2.1(11)
compilation_unit 10.1.1(3)
  used 10.1.1(2), P.1
compile-time error 1.1.2(27), 1.1.5(4)
compile-time semantics 1.1.2(28)
complete context 8.6(4)
completely defined 3.11.1(8)
completion
  abnormal 7.6.1(2/2)
  compile-time concept 3.11.1(1/3)
  normal 7.6.1(2/2)
  run-time concept 7.6.1(2/2)
completion and leaving (completed and left) 7.6.1(2/2)
completion legality
  [*partial*] 3.10.1(13/5)
  entry_body 9.5.2(16)
Complex
  in Ada.Numerics.Generic_Complex_-Types G.1.1(3)
  in Interfaces.Fortran B.5(9)
Complex_Arrays
  child of Ada.Numerics G.3.2(53/2)
Complex_Elementary_Functions
  child of Ada.Numerics G.1.2(9/1)
Complex_IO
  child of Ada.Text_IO G.1.3(3/5)
  child of Ada.Wide_Text_IO G.1.4(1)
  child of Ada.Wide_Wide_Text_IO G.1.5(1/2)
Complex_Matrix
  in Ada.Numerics.Generic_Complex_-Arrays G.3.2(4/2)
Complex_Text_IO
  child of Ada G.1.3(9.1/2)
Complex_Types
  child of Ada.Numerics G.1.1(25/1)
Complex_Vector
  in Ada.Numerics.Generic_Complex_-Arrays G.3.2(4/2)
component 3.2(2/5)
  of a type 3.2(6/2)
  of the nominal subtype 3.9.1(4.2/5)
component subtype 3.6(10)
component_choice_list 4.3.1(5/5)
  used 4.3.1(4/2), P.1
component_clause 13.5.1(3)
  used 13.5.1(2/5), P.1
component_declaration 3.8(6/3)
  used 3.8(5/1), 9.4(6), P.1
component_definition 3.6(7/2)
  used 3.6(3), 3.6(5), 3.8(6/3), P.1
component_item 3.8(5/1)
  used 3.8(4), P.1
component_list 3.8(4)
  used 3.8(3/5), 3.8.1(3), P.1
Component_Size aspect 13.3(70)
Component_Size attribute 13.3(69)
Component_Size clause 13.3(7/2), 13.3(70)
components
  of a record type 3.8(9/2)
Compose
  in Ada.Directories A.16(20/5)
  in Ada.Directories.Hierarchical_File_Names A.16.1(14/3)
Compose attribute A.5.3(24)
Compose_From_Cartesian
  in Ada.Numerics.Generic_Complex_-Arrays G.3.2(9/2), G.3.2(29/2)
  in Ada.Numerics.Generic_Complex_-Types G.1.1(8)
Compose_From_Polar
  in Ada.Numerics.Generic_Complex_-Arrays G.3.2(11/2), G.3.2(32/2)
  in Ada.Numerics.Generic_Complex_-Types G.1.1(11)
composite type 1.3.1(18/5), 3.2(2/5)
composite_constraint 3.2.2(7)
  used 3.2.2(5), P.1
compound delimiter 2.2(10)
compound_statement 5.1(5/5)
  used 5.1(3), P.1
concatenation operator 4.5(1), 4.5.3(3)
concrete subprogram
  See nonabstract subprogram 3.9.3(1/2)
concrete type
  See nonabstract type 3.9.3(1/2)
concurrent
  actions 9.10(16/5)
concurrent processing
  See parallel construct 5.1(1/5)
  See task 9(1/5)
condition 4.5.7(4/3)
  used 4.5.7(3/3), 5.3(2), 5.5(3/5), 5.5(4.1/5), 5.7(2), 9.5.2(7), 9.7.1(3), P.1
  See also exception 11(1/3)
conditional expression 4.5.7(1/3)
conditional_entry_call 9.7.3(2)
  used 9.7(2), P.1
conditional_expression 4.5.7(2/3)
  used 4.4(7/5), P.1
conditionally evaluated
  subexpression 6.1.1(22.16/5)
conditionally executed 5.5(6.3/5)
conditionally produces
  iteration 5.5(6.4/5)
configuration
  of the partitions of a program E(4)
configuration pragma 10.1.5(8)
  Assertion_Policy 11.4.2(7/3)
  Detect_Blocking H.5(4/2)
  Discard_Names C.5(4)
  Locking_Policy D.3(5)
  Normalize_Scalars H.1(4)
  Partition_Elaboration_Policy H.6(5/2)
  Priority_Specific_Dispatching D.2.2(4/2)
  Profile 13.12(14/3)
  Queuing_Policy D.4(5)
  Restrictions 13.12(8/3)
  Reviewable H.3.1(4)
  Suppress 11.5(5/2)
  Task_Dispatching_Policy D.2.2(4/2)
  Unsuppress 11.5(5/2)
confirming
  aspect specification 13.1(18.2/5)
  nonoverridable aspect 13.1.1(18.4/5)
  operational aspect 13.1(18.2/5)
  representation aspect 13.1(18.2/5)
  representation item 13.1(18.2/5)
  representation value 13.1(18.2/5)
conflict 9.10(17/5)
  potentially 9.10(17/5)
conflict check policy 9.10.1(1/5)
Conflict_Check_Policy pragma 9.10.1(3/5), L(6.2/5)
conformance 6.3.1(1)
  of an implementation 1.1.3(1)
  See also full conformance, mode conformance, subtype conformance, type conformance
Conjugate
  in Ada.Numerics.Generic_Complex_-Arrays G.3.2(13/2), G.3.2(34/2)
  in Ada.Numerics.Generic_Complex_-Types G.1.1(12), G.1.1(15)
consistency
  among compilation units 10.1.4(5)
constant 3.3(13/3)
  result of a function_call 6.4(12/2)
  See also literal 4.2(1)
  See also static 4.9(1)
constant indexing 4.1.6(12/3)
constant object 3.3(13/3)
constant view 3.3(13/3)
Constant_Indexing aspect 4.1.6(2/5)
Constant_Reference
  in Ada.Containers.Doubly_Linked_-Lists A.18.3(17.3/5)
  in Ada.Containers.Hashed_Maps A.18.5(17.3/5), A.18.5(17.5/5)
  in Ada.Containers.Hashed_Sets A.18.8(17.3/5), A.18.8(58.3/5)
  in Ada.Containers.Indefinite_Holders A.18.18(18/5)
  in Ada.Containers.Multiway_Trees A.18.10(30/5)
  in Ada.Containers.Ordered_Maps A.18.6(16.3/5), A.18.6(16.5/5)
  in Ada.Containers.Ordered_Sets A.18.9(16.3/5), A.18.9(73.3/5)
  in Ada.Containers.Vectors A.18.2(34.3/5), A.18.2(34.5/5)
Constant_Reference_Type
  in Ada.Containers.Doubly_Linked_-Lists A.18.3(51.10/5)
  in Ada.Containers.Hashed_Maps A.18.5(37.11/5)
  in Ada.Containers.Hashed_Sets A.18.8(59.10/5)
  in Ada.Containers.Indefinite_Holders A.18.18(16/5)

in Ada.Containers.Multiway_Trees A.18.10(70.10/5)
in Ada.Containers.Ordered_Maps A.18.6(51.12/5)
in Ada.Containers.Ordered_Sets A.18.9(74.10/5)
in Ada.Containers.Vectors A.18.2(79.10/5)
Constants
child of Ada.Strings.Maps A.4.6(3/5)
constituent
of a construct 1.1.4(17)
operative 4.4(9.2/5)
constrained 3.2(9)
known to be 3.3(23.1/3)
object 3.3.1(9/2)
object 6.4.1(16)
subtype 3.2(9), 3.4(6), 3.5(7), 3.5.1(10), 3.5.4(9), 3.5.4(10), 3.5.7(11), 3.5.9(13), 3.5.9(16), 3.6(15/5), 3.6(16), 3.7(26), 3.9(15)
subtype 3.10(14/3)
subtype K.2(33)
Constrained attribute 3.7.2(3/5), J.4(2)
constrained by its initial value 3.3.1(9/2)
[*partial*] 4.8(6/3), 6.5(5.12/5)
constrained_array_definition 3.6(5)
used 3.6(2), P.1
constraint 3.2.2(5)
[*partial*] 3.2(7/2)
null 3.2(7/2)
of a first array subtype 3.6(16)
of a subtype 3.2(8/5)
of an object 3.3.1(9/2)
used 3.2.2(3/2), P.1
Constraint_Error
raised by case expression 4.5.7(21/3)
raised by case statement 5.4(13)
raised by detection of a bounded error 13.9.1(9), 13.11.2(15.2/4), A.18.2(238/3), A.18.2(239.2/3), A.18.2(246/2), A.18.3(152.2/3), A.18.4(75.8/3), A.18.7(96.20/3), A.18.10(221/3), A.18.18(69/3)
raised by discriminant-dependent aggregate 4.3.1(19/5)
raised by failure of runtime check 3.2.2(12), 3.5(24), 3.5(27), 3.5(39.12/3), 3.5(39.4/3), 3.5(39.5/3), 3.5(43/5), 3.5(55/5), 3.5.4(20), 3.5.5(7), 3.5.9(19/4), 3.9.2(16), 4.1(13), 4.1.1(7), 4.1.2(7), 4.1.3(15), 4.1.5(8/3), 4.2(11/5), 4.3(6), 4.3.2(8/3), 4.3.3(20.4/5), 4.3.3(31), 4.3.4(16/5), 4.3.4(20/5), 4.4(11), 4.5(10), 4.5(11), 4.5(12), 4.5.1(8), 4.5.3(8), 4.5.5(22), 4.5.6(6), 4.5.6(13), 4.6(28), 4.6(57/5), 4.6(60), 4.7(4/5), 4.8(10.4/3), 4.8(10/2), 5.2(10), 6.5(5.12/5), 6.5(8.2/5), 11.1(4), 11.5(10/5), 13.13.2(35/3), A.5.3(26), A.5.3(29), A.5.3(47), A.5.3(50), A.5.3(53), A.5.3(59), A.5.3(62), E.4(19), E.4(20.u), E.4(20.v), G.1.1(40), G.2.1(12), G.2.2(7), G.2.3(26), G.2.4(3), G.2.6(4), K.2(11), K.2(114), K.2(122), K.2(184), K.2(202), K.2(220), K.2(241), K.2(261), K.2(41), K.2(47)
raised by object of discriminated type 3.8.1(21.1/3)
in Standard A.1(46)
Construct 1.1.4(16), 1.3.3(4/5)
master 7.6.1(3/5)
parallel 5.1(1/5)
constructor
See initialization 3.3.1(18/2)
See initialization 7.6(1)
See initialization expression 3.3.1(4)
See Initialize 7.6(1)
See initialized allocator 4.8(4)
container 1.3.3(5/5)
cursor A.18(2/5)
list A.18.3(1/2)
map A.18.4(1/2)
set A.18.7(1/2)
vector A.18.2(1/2)
container aggregate 1.3.3(7/5), 4.3.5(1/5)
container element iterator 5.5.2(3/3)
container type
aggregate aspect 4.3.5(5/5)
container_aggregate 4.3.5(13/5)
used 4.3(2/5), P.1
container_element_association 4.3.5(18/5)
used 4.3.5(17/5), P.1
container_element_association_list 4.3.5(17/5)
used 4.3.5(16/5), P.1
Containers
child of Ada A.18.1(3/5)
Containers_Assertion_Check 11.5(23.6/5)
Containing_Directory
in Ada.Directories A.16(17/5)
in Ada.Directories.Hierarchical_File_Names A.16.1(11/3)
Contains
in Ada.Containers.Doubly_Linked_Lists A.18.3(43/2)
in Ada.Containers.Hashed_Maps A.18.5(32/2)
in Ada.Containers.Hashed_Sets A.18.8(44/2), A.18.8(57/2)
in Ada.Containers.Multiway_Trees A.18.10(41/3)
in Ada.Containers.Ordered_Maps A.18.6(42/2)
in Ada.Containers.Ordered_Sets A.18.9(52/2), A.18.9(72/2)
in Ada.Containers.Vectors A.18.2(71/2)
context free grammar
complete listing P.1(1/5)
cross reference P.2(1/3)
notation 1.1.4(3)
under Syntax heading 1.1.2(25)
context_clause 10.1.2(2)
used 10.1.1(3), P.1
context_item 10.1.2(3)
used 10.1.2(2), P.1
contiguous representation
[*partial*] 13.1(7.a/2), 13.5.2(5), 13.7.1(12), 13.9(9), 13.9(17/3), 13.11(17.d), 13.11(21.6/5)
Continue
in Ada.Asynchronous_Task_Control D.11(3/5)
contract model of generics 12.3(1.a/5)
control character
a category of Character A.3.2(22)
a category of Character A.3.3(4), A.3.3(15)
See also format_effector 2.1(13/3)
Control_Set
in Ada.Strings.Maps.Constants A.4.6(4)
Controlled
in Ada.Finalization 7.6(5/5)
controlled type 1.3.1(19/5), 7.6(2), 7.6(9/2)
controlling access result 3.9.2(2/3)
controlling formal parameter 3.9.2(2/3)
controlling operand 3.9.2(2/3)
controlling result 3.9.2(2/3)
controlling tag
for a call on a dispatching operation 3.9.2(1/5)
controlling tag value 3.9.2(14)
for the expression in an assignment_statement 5.2(9)
controlling type
of a formal_abstract_subprogram_declaration 12.6(8.5/5)
convention 6.3.1(2/1), B.1(11/3)
Convention aspect B.1(2/3)
Convention pragma J.15.5(4/3), L(8.1/3)
conversion 4.6(1/3), 4.6(28)
access 4.6(24.11/2), 4.6(24.18/2), 4.6(24.19/2), 4.6(47)

arbitrary order 1.1.4(18)
array 4.6(24.2/2), 4.6(36)
composite (non-array) 4.6(21/3), 4.6(40)
enumeration 4.6(21.1/2), 4.6(34)
numeric 4.6(24.1/2), 4.6(29)
unchecked 13.9(1)
value 4.6(5/5)
view 4.6(5/5)
Conversion_Error
in Interfaces.COBOL B.4(30)
Conversions
child of Ada.Characters A.3.4(2/5)
child of Ada.Strings.UTF_Encoding A.4.11(15/5)
Convert
in Ada.Strings.UTF_Encoding.Conversions A.4.11(16/3), A.4.11(17/3), A.4.11(18/3), A.4.11(19/3), A.4.11(20/3)
convertible 4.6(4/3)
required 4.6(24.13/2), 4.6(24.4/2), 8.6(27.1/4)
Copy
in Ada.Containers.Bounded_Doubly_Linked_Lists A.18.20(10/5)
in Ada.Containers.Bounded_Hashed_Maps A.18.21(13/5)
in Ada.Containers.Bounded_Hashed_Sets A.18.23(13/5)
in Ada.Containers.Bounded_Ordered_Maps A.18.22(10/5)
in Ada.Containers.Bounded_Ordered_Sets A.18.24(10/5)
in Ada.Containers.Doubly_Linked_Lists A.18.3(17.6/5), A.18.3(51.9/5)
in Ada.Containers.Hashed_Maps A.18.5(17.8/5), A.18.5(37.10/5)
in Ada.Containers.Hashed_Sets A.18.8(17.5/5), A.18.8(59.9/5)
in Ada.Containers.Indefinite_Holders A.18.18(21/5)
in Ada.Containers.Multiway_Trees A.18.10(33/5), A.18.10(70.9/5)
in Ada.Containers.Ordered_Maps A.18.6(16.8/5), A.18.6(51.11/5)
in Ada.Containers.Ordered_Sets A.18.9(16.5/5), A.18.9(74.9/5)
in Ada.Containers.Vectors A.18.2(34.8/5), A.18.2(79.9/5)
copy back of parameters 6.4.1(17)
copy parameter passing 6.2(2)
Copy_Array
in Interfaces.C.Pointers B.3.2(15)
Copy_File
in Ada.Directories A.16(13/2)
Copy_Local_Subtree
in Ada.Containers.Multiway_Trees A.18.10(54.1/5)
Copy_Sign attribute A.5.3(51)
Copy_Subtree
in Ada.Containers.Multiway_Trees A.18.10(54.2/5), A.18.10(54/5)
Copy_Terminated_Array
in Interfaces.C.Pointers B.3.2(14)
Copyright_Sign
in Ada.Characters.Latin_1 A.3.3(21/3)
core language 1.1.2(2), 1.3.3(8/5)
Correction 1.1.2(39.aa/3), 1.1.2(39.nn/5), 1.1.2(39.qq/5)
corresponding constraint 3.4(6)
corresponding discriminants 3.7(18)
corresponding expression
class-wide type invariant 7.3.2(8/5)
corresponding expresssion
class-wide postcondition 6.1.1(18/5)
class-wide precondition 6.1.1(18/5)
corresponding index
for an array_aggregate 4.3.3(8)
corresponding subtype 3.4(18/3)
corresponding value
of the target type of a conversion 4.6(28)
Corrigendum 1.1.2(39.n/3), 1.1.2(39.nn/5), 1.1.2(39.qq/5), 1.1.2(39.tt/5), 1.1.2(39.ww/5)
Cos
in Ada.Numerics.Generic_Complex_Elementary_Functions G.1.2(4)
in Ada.Numerics.Generic_Elementary_Functions A.5.1(5)
Cosh
in Ada.Numerics.Generic_Complex_Elementary_Functions G.1.2(6)
in Ada.Numerics.Generic_Elementary_Functions A.5.1(7)
Cot
in Ada.Numerics.Generic_Complex_Elementary_Functions G.1.2(4)
in Ada.Numerics.Generic_Elementary_Functions A.5.1(5)
Coth
in Ada.Numerics.Generic_Complex_Elementary_Functions G.1.2(6)
in Ada.Numerics.Generic_Elementary_Functions A.5.1(7)
Count
in Ada.Direct_IO A.8.4(4)
in Ada.Streams.Stream_IO A.12.1(7)
in Ada.Strings.Bounded A.4.4(48), A.4.4(49), A.4.4(50)
in Ada.Strings.Fixed A.4.3(13), A.4.3(14), A.4.3(15)
in Ada.Strings.Unbounded A.4.5(43), A.4.5(44), A.4.5(45)
in Ada.Text_IO A.10.1(5)
Count attribute 9.9(5)
Count_Type
in Ada.Containers A.18.1(5/2)
Country
in Ada.Locales A.19(6/3)
Country code standard 1.2(4.1/5)
Country_Code
in Ada.Locales A.19(4/4)
Country_Unknown
in Ada.Locales A.19(5/3)
cover
a type 3.4.1(9/5)
of a choice and an exception 11.2(6)
cover a value 3.8.1(1.a)
by a discrete_choice 3.8.1(9)
by a discrete_choice_list 3.8.1(13)
covered
global aspect 6.1.2(38/5)
CPU aspect D.16(8/3)
CPU clock tick D.14(15/2)
CPU pragma J.15.9(2/3), L(8.2/3)
CPU *subtype of* CPU_Range
in System.Multiprocessors D.16(4/3)
CPU time
of a task D.14(11/3)
CPU_Range
in System.Multiprocessors D.16(4/3)
CPU_Set
in System.Multiprocessors.Dispatching_Domains D.16.1(9.1/4)
CPU_Tick
in Ada.Execution_Time D.14(4/2)
CPU_Time
in Ada.Execution_Time D.14(4/2)
CPU_Time_First
in Ada.Execution_Time D.14(4/2)
CPU_Time_Last
in Ada.Execution_Time D.14(4/2)
CPU_Time_Unit
in Ada.Execution_Time D.14(4/2)
CR
in Ada.Characters.Latin_1 A.3.3(5)
create 3.1(12)
in Ada.Direct_IO A.8.4(6), A.8.4(18.10/5), A.8.4(18.3/5)
in Ada.Sequential_IO A.8.1(6), A.8.1(15.10/5), A.8.1(15.3/5)
in Ada.Streams.Stream_IO A.12.1(8), A.12.1(26.10/5), A.12.1(26.3/5)
in Ada.Text_IO A.10.1(9), A.10.1(85.10/5), A.10.1(85.3/5)
in System.Multiprocessors.Dispatching_Domains D.16.1(7/4), D.16.1(9.2/4)
Create_Directory
in Ada.Directories A.16(7/2)
Create_Path
in Ada.Directories A.16(9/2)
Create_Subpool
in System.Storage_Pools.Subpools 13.11.4(7/3)
creation
of a protected object C.3.1(10/3)
of a return object 6.5(5.12/5)
of a tag 13.14(20/2)
of a task object D.1(17/4)

of an object 3.3(1)
critical section
See intertask communication 9.5(1)
CSI
in Ada.Characters.Latin_1 A.3.3(19)
Currency_Sign
in Ada.Characters.Latin_1 A.3.3(21/3)
current column number A.10(9)
current index
of an open direct file A.8(4)
of an open stream file A.12.1(1.1/1)
current instance
of a generic unit 8.6(18)
of a type 8.6(17/3)
current line number A.10(9)
current mode
of an open file A.7(7)
current page number A.10(9)
Current size
of a stream file A.12.1(1.1/1)
of an external file A.8(3)
Current_Directory
in Ada.Directories A.16(5/2)
Current_Error
in Ada.Text_IO A.10.1(17), A.10.1(20)
Current_Handler
in Ada.Execution_Time.Group_Budgets D.14.2(10/2)
in Ada.Execution_Time.Timers D.14.1(7/2)
in Ada.Interrupts C.3.2(6)
in Ada.Real_Time.Timing_Events D.15(5/2)
Current_Indent
in Ada.Strings.Text_Buffers A.4.12(14/5)
Current_Input
in Ada.Text_IO A.10.1(17), A.10.1(20)
Current_Output
in Ada.Text_IO A.10.1(17), A.10.1(20)
Current_State
in Ada.Synchronous_Task_Control D.10(4/5)
Current_Task
in Ada.Task_Identification C.7.1(3/5)
Current_Task_Fallback_Handler
in Ada.Task_Termination C.7.3(5/2)
Current_Use
in Ada.Containers.Bounded_Priority_Queues A.18.31(7/5)
in Ada.Containers.Bounded_Synchronized_Queues A.18.29(6/5)
in Ada.Containers.Synchronized_Queue_Interfaces A.18.27(7/5)
in Ada.Containers.Unbounded_Priority_Queues A.18.30(7/5)
in Ada.Containers.Unbounded_Synchronized_Queues A.18.28(6/5)
cursor
ambiguous A.18.2(240/2)
for a container A.18(2/5)
invalid A.18.2(248/2), A.18.3(153/2), A.18.4(76/2), A.18.7(97/2), A.18.10(222/3)
in Ada.Containers.Doubly_Linked_Lists A.18.3(7/5), A.18.3(51.3/5)
in Ada.Containers.Hashed_Maps A.18.5(4/5), A.18.5(37.4/5)
in Ada.Containers.Hashed_Sets A.18.8(4/5), A.18.8(59.3/5)
in Ada.Containers.Multiway_Trees A.18.10(9/5), A.18.10(70.3/5)
in Ada.Containers.Ordered_Maps A.18.6(5/5), A.18.6(51.5/5)
in Ada.Containers.Ordered_Sets A.18.9(5/5), A.18.9(74.3/5)
in Ada.Containers.Vectors A.18.2(9/5), A.18.2(79.3/5)

D

dangling reference 13.11.2(15.1/4)
dangling references
prevention via accessibility rules 3.10.2(3/5)
Data_Error
in Ada.Direct_IO A.8.4(18)
in Ada.IO_Exceptions A.13(4)
in Ada.Sequential_IO A.8.1(15)
in Ada.Storage_IO A.9(9)
in Ada.Streams.Stream_IO A.12.1(26)
in Ada.Text_IO A.10.1(85/5)
date and time formatting standard 1.2.1(6/5)
Day
in Ada.Calendar 9.6(13)
in Ada.Calendar.Formatting 9.6.1(23/2)
Day_Count
in Ada.Calendar.Arithmetic 9.6.1(10/2)
Day_Duration *subtype of* Duration
in Ada.Calendar 9.6(11/2)
Day_Name
in Ada.Calendar.Formatting 9.6.1(17/2)
Day_Number *subtype of* Integer
in Ada.Calendar 9.6(11/2)
Day_of_Week
in Ada.Calendar.Formatting 9.6.1(18/2)
DC1
in Ada.Characters.Latin_1 A.3.3(6)
DC2
in Ada.Characters.Latin_1 A.3.3(6)
DC3
in Ada.Characters.Latin_1 A.3.3(6)
DC4
in Ada.Characters.Latin_1 A.3.3(6)
DCS
in Ada.Characters.Latin_1 A.3.3(18)
deadline
active D.2.6(14/5)
base D.2.6(14/5)
relative D.3(13.2/5)
Deadline *subtype of* Time
in Ada.Dispatching.EDF D.2.6(9/5)
Deallocate
in System.Storage_Pools 13.11(8)
in System.Storage_Pools.Subpools 13.11.4(15/3)
Deallocate_Subpool
in System.Storage_Pools.Subpools 13.11.4(12/3)
deallocation of storage 13.11.2(1)
Decimal
child of Ada F.2(2/5)
decimal digit
a category of Character A.3.2(28)
decimal fixed point type 3.5.9(1), 3.5.9(6)
Decimal_Conversions
in Interfaces.COBOL B.4(31)
Decimal_Digit_Set
in Ada.Strings.Maps.Constants A.4.6(4)
Decimal_Element
in Interfaces.COBOL B.4(12/3)
decimal_fixed_point_definition 3.5.9(4)
used 3.5.9(2), P.1
Decimal_IO
in Ada.Text_IO A.10.1(73)
decimal_literal 2.4.1(2)
used 2.4(2), P.1
Decimal_Output
in Ada.Text_IO.Editing F.3.3(11)
declaration 1.3.3(10/5), 3.1(5), 3.1(6/5)
declaration list
[*partial*] 12.1(9.1/5)
declarative_part 3.11(6.1/2)
package_specification 7.1(6/2)
declarative region
of a construct 8.1(1)
declarative_item 3.11(3)
used 3.11(2), P.1
declarative_part 3.11(2)
used 5.6(2), 6.3(2/3), 7.2(2/3), 9.1(6/3), 9.5.2(5/5), P.1
declare 3.1(8), 3.1(12)
declare expression 4.5.9(1/5)
declare_expression 4.5.9(2/5)
used 4.4(7/5), P.1
declare_item 4.5.9(3/5)
used 4.5.9(2/5), P.1
declared pure 10.2.1(17/5)
Decode
in Ada.Strings.UTF_Encoding.Strings A.4.11(26/3), A.4.11(27/3), A.4.11(28/3)
in Ada.Strings.UTF_Encoding.Wide_Strings A.4.11(34/3), A.4.11(35/3), A.4.11(36/3)
in Ada.Strings.UTF_Encoding.Wide_Wide_Strings A.4.11(42/3), A.4.11(43/3), A.4.11(44/3)

Decrease_Indent
  in Ada.Strings.Text_Buffers A.4.12(16/5)
Decrement
  in Interfaces.C.Pointers B.3.2(11/3)
deeper
  accessibility level 3.10.2(3/5)
  statically 3.10.2(4), 3.10.2(17)
default constant indexing function 5.5.1(16/3)
default cursor subtype 5.5.1(8/5)
default directory A.16(48/2)
default element subtype 5.5.1(9/5)
default entry queuing policy 9.5.3(17)
default initial condition 1.3.1(21/5)
default initial condition check 7.3.3(9/5)
default initial condition expression 7.3.3(2/5)
default iterator function 5.5.1(8/5)
default iterator subtype 5.5.1(8/5)
default pool 13.11.3(4.2/4)
default treatment C.3(5)
default variable indexing function 5.5.1(21/3)
Default_Aft
  in Ada.Text_IO A.10.1(64), A.10.1(69), A.10.1(74)
  in Ada.Text_IO.Complex_IO G.1.3(5)
Default_Base
  in Ada.Text_IO A.10.1(53), A.10.1(58)
Default_Bit_Order
  in System 13.7(15/2)
Default_Component_Value aspect 3.6(22.2/3)
Default_Currency
  in Ada.Text_IO.Editing F.3.3(10)
Default_Deadline
  in Ada.Dispatching.EDF D.2.6(9/5)
Default_Exp
  in Ada.Text_IO A.10.1(64), A.10.1(69), A.10.1(74)
  in Ada.Text_IO.Complex_IO G.1.3(5)
default_expression 3.7(6)
  used 3.7(5/5), 3.8(6/3), 6.1(15/5), 12.4(2/3), P.1
Default_Fill
  in Ada.Text_IO.Editing F.3.3(10)
Default_Fore
  in Ada.Text_IO A.10.1(64), A.10.1(69), A.10.1(74)
  in Ada.Text_IO.Complex_IO G.1.3(5)
Default_Initial_Condition aspect 7.3.3(2/5)
Default_Iterator aspect 5.5.1(8/5)
Default_Modulus
  in Ada.Containers.Bounded_Hashed_Maps A.18.21(10/3)
  in Ada.Containers.Bounded_Hashed_Sets A.18.23(10/3)
default_name 12.6(4)
  used 12.6(3/2), P.1
Default_Priority
  in System 13.7(17)
Default_Quantum
  in Ada.Dispatching.Round_Robin D.2.5(4/5)
Default_Radix_Mark
  in Ada.Text_IO.Editing F.3.3(10)
Default_Relative_Deadline
  in Ada.Dispatching.EDF D.2.6(9/5)
Default_Separator
  in Ada.Text_IO.Editing F.3.3(10)
Default_Setting
  in Ada.Text_IO A.10.1(80)
Default_Storage_Pool aspect 13.11.3(5/4)
Default_Storage_Pool pragma 13.11.3(3/3), L(8.3/3)
Default_Subpool_for_Pool
  in System.Storage_Pools.Subpools 13.11.4(13/3)
Default_Value aspect 3.5(56.3/3)
Default_Width
  in Ada.Text_IO A.10.1(53), A.10.1(58), A.10.1(80)
deferred constant 7.4(2/3)
deferred constant declaration 3.3.1(6/5), 7.4(2/3)
defining name 3.1(10)
defining_character_literal 3.5.1(4)
  used 3.5.1(3), P.1
defining_designator 6.1(6)
  used 6.1(4.2/2), 12.3(2/3), P.1
defining_identifier 3.1(4)
  used 3.2.1(3/3), 3.2.2(2/3), 3.3.1(3), 3.5.1(3), 3.10.1(2/2), 4.3.3(5.1/5), 5.5(3.1/5), 5.5(4/5), 5.5.2(2/5), 5.5.3(3/5), 6.1(7), 6.5(2.1/5), 7.3(2/3), 7.3(3/3), 8.5.1(2/5), 8.5.2(2/3), 9.1(2/3), 9.1(3/5), 9.1(6/3), 9.4(2/3), 9.4(3/3), 9.4(7/3), 9.5.2(2/3), 9.5.2(5/5), 9.5.2(8/5), 10.1.3(4/3), 10.1.3(5/3), 10.1.3(6/3), 11.2(4), 12.5(2.1/5), 12.5(2.2/5), 12.7(2/3), P.1
defining_identifier_list 3.3.1(3)
  used 3.3.1(2/3), 3.3.2(2), 3.7(5/5), 3.8(6/3), 6.1(15/5), 11.1(2/3), 12.4(2/3), P.1
defining_operator_symbol 6.1(11)
  used 6.1(6), P.1
defining_program_unit_name 6.1(7)
  used 6.1(4.1/2), 6.1(6), 7.1(3/3), 7.2(2/3), 8.5.3(2/3), 8.5.5(2/3), 12.3(2/3), P.1
Definite attribute 12.5.1(23/3)
definite subtype 3.3(23/5)
definition 3.1(7)
Degree_Sign
  in Ada.Characters.Latin_1 A.3.3(22)
DEL
  in Ada.Characters.Latin_1 A.3.3(14)
delay_alternative 9.7.1(6)
  used 9.7.1(4), 9.7.2(2), P.1
delay_relative_statement 9.6(4)
  used 9.6(2), P.1
delay_statement 9.6(2)
  used 5.1(4/2), 9.7.1(6), 9.7.4(4/2), P.1
Delay_Until_And_Set_CPU
  in System.Multiprocessors.Dispatching_Domains D.16.1(14/3)
Delay_Until_And_Set_Deadline
  in Ada.Dispatching.EDF D.2.6(9/5)
delay_until_statement 9.6(3)
  used 9.6(2), P.1
Delete
  in Ada.Containers.Doubly_Linked_Lists A.18.3(24/5)
  in Ada.Containers.Hashed_Maps A.18.5(25/5), A.18.5(26/5)
  in Ada.Containers.Hashed_Sets A.18.8(24/5), A.18.8(25/5), A.18.8(55/5)
  in Ada.Containers.Ordered_Maps A.18.6(24/5), A.18.6(25/5)
  in Ada.Containers.Ordered_Sets A.18.9(23/5), A.18.9(24/5), A.18.9(68/5)
  in Ada.Containers.Vectors A.18.2(50/5), A.18.2(51/5)
  in Ada.Direct_IO A.8.4(8)
  in Ada.Sequential_IO A.8.1(8)
  in Ada.Streams.Stream_IO A.12.1(10)
  in Ada.Strings.Bounded A.4.4(64), A.4.4(65)
  in Ada.Strings.Fixed A.4.3(29), A.4.3(30)
  in Ada.Strings.Unbounded A.4.5(59), A.4.5(60)
  in Ada.Text_IO A.10.1(11)
Delete_Children
  in Ada.Containers.Multiway_Trees A.18.10(53/5)
Delete_Directory
  in Ada.Directories A.16(8/2)
Delete_File
  in Ada.Directories A.16(11/2)
Delete_First
  in Ada.Containers.Doubly_Linked_Lists A.18.3(25/5)
  in Ada.Containers.Ordered_Maps A.18.6(26/5)
  in Ada.Containers.Ordered_Sets A.18.9(25/5)
  in Ada.Containers.Vectors A.18.2(52/5)
Delete_Last
  in Ada.Containers.Doubly_Linked_Lists A.18.3(26/5)
  in Ada.Containers.Ordered_Maps A.18.6(27/5)
  in Ada.Containers.Ordered_Sets A.18.9(26/5)
  in Ada.Containers.Vectors A.18.2(53/5)
Delete_Leaf
  in Ada.Containers.Multiway_Trees A.18.10(35/5)

Delete_Subtree
  in Ada.Containers.Multiway_Trees A.18.10(36/5)
Delete_Tree
  in Ada.Directories A.16(10/2)
delimiter 2.2(8/2)
delivery
  of an interrupt C.3(2)
delta
  of a fixed point type 3.5.9(1)
delta aggregate 4.3.4(1/5)
Delta attribute 3.5.10(3)
delta_aggregate 4.3.4(2/5)
  used 4.3(2/5), P.1
delta_constraint J.3(2/4)
  used 3.2.2(6), P.1
Denominator
  in Ada.Numerics.Big_Numbers.Big_Reals A.5.7(8/5)
Denorm attribute A.5.3(9)
denormalized number A.5.3(10)
denote 8.6(16)
  informal definition 3.1(8)
  name used as a pragma argument 8.6(32)
depend on a discriminant
  for a component 3.7(20)
  for a constraint or component_definition 3.7(19)
dependence
  elaboration 10.2(9/5)
  of a task on a master 9.3(1)
  of a task on another task 9.3(4)
  semantic 10.1.1(26/2)
depth
  accessibility level 3.10.2(3/5)
  in Ada.Containers.Multiway_Trees A.18.10(19.1/5), A.18.10(19/5)
depth-first order A.18.10(5/3)
Dequeue
  in Ada.Containers.Bounded_Priority_Queues A.18.31(5/3)
  in Ada.Containers.Bounded_Synchronized_Queues A.18.29(5/3)
  in Ada.Containers.Synchronized_Queue_Interfaces A.18.27(6/5)
  in Ada.Containers.Unbounded_Priority_Queues A.18.30(5/3)
  in Ada.Containers.Unbounded_Synchronized_Queues A.18.28(5/3)
Dequeue_Only_High_Priority
  in Ada.Containers.Bounded_Priority_Queues A.18.31(6/3)
  in Ada.Containers.Unbounded_Priority_Queues A.18.30(6/3)
dereference 4.1(8)
Dereference_Error
  in Interfaces.C.Strings B.3.1(12)
derivation class
  for a type 3.4.1(2/2)
derived from
  directly or indirectly 3.4.1(2/2)
derived type 1.3.1(22/5), 3.4(1/2)
  [*partial*] 3.4(24)
derived_type_definition 3.4(2/2)
  used 3.2.1(4/2), P.1
descendant 10.1.1(11)
  at run time 3.9(12.3/3)
  of a tree node A.18.10(4/3)
  of a type 3.4.1(10/2)
  of the full view of a type 7.3.1(5.2/5)
  relationship with scope 8.2(4)
descendant of a type 1.3.1(25/5)
Descendant_Tag
  in Ada.Tags 3.9(7.1/2)
designate 3.10(1)
designated profile
  of an access-to-subprogram type 3.10(11)
  of an anonymous access type 3.10(12/3)
designated subtype
  of a named access type 3.10(10)
  of an anonymous access type 3.10(12/3)
designated type
  of a named access type 3.10(10)
  of an anonymous access type 3.10(12/3)
designator 6.1(5)
  used 6.3(2/3), P.1
destructor
  See finalization 7.6(1)
  See finalization 7.6.1(1)
Detach_Handler
  in Ada.Interrupts C.3.2(9)
Detect_Blocking pragma H.5(3/2), L(8.4/2)
Determinant
  in Ada.Numerics.Generic_Complex_-Arrays G.3.2(46/2)
  in Ada.Numerics.Generic_Real_Arrays G.3.1(24/2)
determined category for a formal type 12.5(6/3)
determined to be unevaluated 6.1.1(24.1/5)
determines
  a type by a subtype_mark 3.2.2(8)
determining expression 6.1.1(22.16/5)
Device_Error
  in Ada.Direct_IO A.8.4(18)
  in Ada.Directories A.16(43/2)
  in Ada.IO_Exceptions A.13(4)
  in Ada.Sequential_IO A.8.1(15)
  in Ada.Streams.Stream_IO A.12.1(26)
  in Ada.Text_IO A.10.1(85/5)
Diaeresis
  in Ada.Characters.Latin_1 A.3.3(21/3)
Difference
  in Ada.Calendar.Arithmetic 9.6.1(12/2)
  in Ada.Containers.Hashed_Sets A.18.8(32/5), A.18.8(33/5)
  in Ada.Containers.Ordered_Sets A.18.9(33/5), A.18.9(34/5)
digit 2.4.1(4.1/2)
  used 2.4.1(3), 2.4.2(5), P.1
digits
  of a decimal fixed point subtype 3.5.9(6), 3.5.10(7)
Digits attribute 3.5.8(2/1), 3.5.10(7)
digits_constraint 3.5.9(5/4)
  used 3.2.2(6), P.1
dimensionality
  of an array 3.6(12)
direct access A.8(3)
direct file A.8(1/2)
Direct_IO
  child of Ada A.8.4(2/5)
direct_name 4.1(3)
  used 3.8.1(2), 4.1(2/5), 5.1(8), 9.5.2(3), 13.1(3), J.7(1), J.15.14(4/5), L(25.3/5), P.1
Direction
  in Ada.Strings A.4.1(6)
directly specified
  of a representation aspect of an entity 13.1(8/5)
  of an operational aspect of an entity 13.1(8.1/3)
directly visible 8.3(2), 8.3(21)
  within a pragma in a context_clause 10.1.6(3)
  within a pragma that appears at the place of a compilation unit 10.1.6(5)
  within a use_clause in a context_clause 10.1.6(3)
  within a with_clause 10.1.6(2/2)
  within the parent_unit_name of a library unit 10.1.6(2/2)
  within the parent_unit_name of a subunit 10.1.6(4)
Directories
  child of Ada A.16(3/5)
directory A.16(45/2)
directory entry A.16(49/2)
directory name A.16(46/2)
Directory_Entry_Type
  in Ada.Directories A.16(29/2)
disabled
  predicate checks 3.2.4(7/3)
Discard_Names aspect C.5(1.3/4)
Discard_Names pragma C.5(3), L(9)
discontiguous representation
  [*partial*] 13.1(7.a/2), 13.5.2(5), 13.7.1(12), 13.9(9), 13.9(17/3), 13.11(17.d), 13.11(21.6/5)
discrete array type 4.5.2(1)
discrete type 1.3.1(27/5), 3.2(3), 3.5(1)
discrete_choice 3.8.1(5/3)
  used 3.8.1(4/5), P.1
discrete_choice_list 3.8.1(4/5)

used 3.8.1(3), 4.3.3(5.1/5), 4.3.3(5/5), 4.5.7(6/3), 5.4(3), P.1
Discrete_Random
child of Ada.Numerics A.5.2(17/5)
discrete_range 3.6.1(3)
used 3.6.1(2), 4.1.2(2), 4.3.5(20/5), P.1
discrete_subtype_definition 3.6(6)
used 3.6(5), 5.5(3.1/5), 5.5(4/5), 9.5.2(2/3), 9.5.2(8/5), P.1
discriminant 1.3.1(28/5), 3.2(5/2), 3.7(1/2)
of a variant_part 3.8.1(6)
use in a record definition 3.8(12/3)
discriminant_association 3.7.1(3/5)
used 3.7.1(2), P.1
Discriminant_Check 11.5(12)
[*partial*] 4.1.3(15), 4.3(6), 4.3.2(8/3), 4.3.4(16/5), 4.6(43), 4.6(45), 4.6(51/5), 4.6(52), 4.7(4/5), 4.8(10/2), 6.5(5.12/5)
discriminant_constraint 3.7.1(2)
used 3.2.2(7), P.1
discriminant_part 3.7(2)
used 3.10.1(2/2), 7.3(2/3), 7.3(3/3), 12.5(2.1/5), 12.5(2.2/5), P.1
discriminant_specification 3.7(5/5)
used 3.7(4), P.1
discriminants
known 3.7(26)
unknown 3.7(1.b/2)
unknown 3.7(26)
discriminated type 3.7(8/2)
dispatching 3.9(3)
child of Ada D.2.1(1.2/5)
Dispatching aspect H.7.1(8/5)
dispatching call
on a dispatching operation 3.9.2(1/5)
dispatching domain D.16.1(16/4)
dispatching operation 3.9.2(1/5), 3.9.2(2/3)
[*partial*] 3.9(1)
dispatching operation set H.7.1(14/5)
dispatching point D.2.1(4/5)
[*partial*] D.2.3(8/2), D.2.4(9/3)
dispatching policy for tasks 9(10.a/3)
[*partial*] D.2.1(5/2)
dispatching, task D.2.1(4/5)
Dispatching_Domain
in System.Multiprocessors.Dispatching_Domains D.16.1(5/3)
Dispatching_Domain aspect D.16.1(18/3)
Dispatching_Domain pragma J.15.10(2/3), L(9.1/3)
Dispatching_Domain_Error
in System.Multiprocessors.Dispatching_Domains D.16.1(4/3)
Dispatching_Domains
child of System.Multiprocessors D.16.1(3/5)
dispatching_operation_set H.7.1(9/5)
dispatching_operation_specifier H.7.1(10/5)
used H.7.1(9/5), P.1
Dispatching_Policy_Error
in Ada.Dispatching D.2.1(1.4/3)
Display_Format
in Interfaces.COBOL B.4(22)
displayed magnitude (of a decimal value) F.3.2(14)
disruption of an assignment 9.8(21), 13.9.1(5)
[*partial*] 11.6(6/5)
distinct access paths 6.2(12/3)
distinguished receiver notation 4.1.3(19.e/2)
distributed accessibility 3.10.2(32.1/3)
distributed program E(3)
distributed system E(2)
distributed systems C(1)
divide 2.1(15/5)
in Ada.Decimal F.2(6/3)
divide operator 4.5(1), 4.5.5(1)
Division_Check 11.5(13/2)
[*partial*] 3.5.4(20), 4.5.5(22), A.5.3(47), G.1.1(40), K.2(202)
Division_Sign
in Ada.Characters.Latin_1 A.3.3(26)
DLE
in Ada.Characters.Latin_1 A.3.3(6)
Do_APC
in System.RPC E.5(10)
Do_RPC
in System.RPC E.5(9)
documentation (required of an implementation) 1.1.3(18), M.1(1/5), M.2(1/5), M.3(1/5)
documentation requirements 1.1.2(34), M(1/5)
summary of requirements M.1(1/5)
Dollar_Sign
in Ada.Characters.Latin_1 A.3.3(8)
dope 13.5.1(15.d)
dot 2.1(15/5)
dot selection
See selected_component 4.1.3(1)
double
in Interfaces.C B.3(16)
Double_Complex
in Interfaces.Fortran B.5(10.2/5)
Double_Imaginary *subtype of* Imaginary
in Interfaces.Fortran B.5(10.3/5)
Double_Precision
in Interfaces.Fortran B.5(6)
Double_Precision_Complex_Types
in Interfaces.Fortran B.5(10.1/5)
Doubly_Linked_Lists
child of Ada.Containers A.18.3(5/5)
downward closure 3.10.2(13.b/2), 3.10.2(37/5)
drift rate D.8(41)
Duration
in Standard A.1(43)
dynamic binding
See dispatching operation 3.9(1)
dynamic semantics 1.1.2(30)
Dynamic_Predicate aspect 3.2.4(1/5)
Dynamic_Priorities
child of Ada D.5.1(3/5)
dynamically determined tag 3.9.2(1/5)
dynamically enclosing
of one execution by another 11.4(2)
dynamically tagged 3.9.2(5/2)

E

e
in Ada.Numerics A.5(3/5)
EDF
child of Ada.Dispatching D.2.6(9/5)
child of Ada.Synchronous_Task_Control D.10(5.2/5)
EDF_Within_Priorities task dispatching policy D.2.6(7/5)
edited output F.3(1/2)
Editing
child of Ada.Text_IO F.3.3(3/5)
child of Ada.Wide_Text_IO F.3.4(1)
child of Ada.Wide_Wide_Text_IO F.3.5(1/2)
effect
external 1.1.3(8)
efficiency 11.5(29/5), 11.6(1/3)
Eigensystem
in Ada.Numerics.Generic_Complex_Arrays G.3.2(49/2)
in Ada.Numerics.Generic_Real_Arrays G.3.1(27/2)
Eigenvalues
in Ada.Numerics.Generic_Complex_Arrays G.3.2(48/2)
in Ada.Numerics.Generic_Real_Arrays G.3.1(26/2)
elaborable 3.1(11.f)
Elaborate pragma 10.2.1(20), L(10)
Elaborate_All pragma 10.2.1(21), L(11)
Elaborate_Body aspect 10.2.1(26.1/5)
Elaborate_Body pragma J.15.14(6/5), L(12.1/5)
elaborated 3.11(8)
elaboration 1.3.4(3/5), 3.1(11)
abstract_subprogram_declaration 3.9.3(11.1/2)
access_definition 3.10(17/2)
access_type_definition 3.10(16)
array_type_definition 3.6(21)
aspect_clause 13.1(19/1)
choice_parameter_specification 11.4(7)
chunk_specification 5.5(8.1/5)
component_declaration 3.8(17)
component_definition 3.6(22/2), 3.8(18/2)
component_list 3.8(17)
declaration with a True Import aspect B.1(38/3)
declarative_part 3.11(7)
deferred constant declaration 7.4(10/3)
delta_constraint J.3(11)
derived_type_definition 3.4(26)
digits_constraint 3.5.9(19/4)

discrete_subtype_definition 3.6(22/2)
discriminant_constraint 3.7.1(12/5)
entry_declaration 9.5.2(22/1)
enumeration_type_definition 3.5.1(10)
exception_declaration 11.1(5)
expression_function_declaration 6.8(8/3)
fixed_point_definition 3.5.9(17)
floating_point_definition 3.5.7(13)
full type definition 3.2.1(11)
full_type_declaration 3.2.1(11)
generic body 12.2(2)
generic_declaration 12.1(10)
generic_instantiation 12.3(20)
incomplete_type_declaration 3.10.1(12)
index_constraint 3.6.1(8)
integer_type_definition 3.5.4(18)
loop_parameter_specification 5.5(9/5)
nongeneric package_body 7.2(6)
nongeneric subprogram_body 6.3(6)
null_procedure_declaration 6.7(5/3)
number_declaration 3.3.2(7)
object_declaration 3.3.1(15)
of library units for a foreign language main subprogram B.1(39/3)
package_body of Standard A.1(50)
package_declaration 7.1(8)
partition E.1(6)
partition E.5(21)
per-object constraint 3.8(18.1/1)
pragma 2.8(12)
private_extension_declaration 7.3(17)
private_type_declaration 7.3(17)
protected declaration 9.4(12)
protected_body 9.4(15)
protected_definition 9.4(13)
range_constraint 3.5(9)
real_type_definition 3.5.6(5)
record_definition 3.8(16)
record_extension_part 3.9.1(5)
record_type_definition 3.8(16)
renaming_declaration 8.5(3)
single_protected_declaration 9.4(12)
single_task_declaration 9.1(10)
subprogram_declaration 6.1(31/2)
subtype_declaration 3.2.2(9)
subtype_indication 3.2.2(9)
task declaration 9.1(10)
task_body 9.1(13)
task_definition 9.1(11)
use_clause 8.4(12)
variant_part 3.8.1(22)
elaboration control 10.2.1(1/5)
elaboration dependence
library_item on another 10.2(9/5)
Elaboration_Check 11.5(20)
[*partial*] 3.11(9)
element
of a storage pool 13.11(11)
in Ada.Containers.Doubly_Linked_Lists A.18.3(14.1/5), A.18.3(14/5)
in Ada.Containers.Hashed_Maps A.18.5(14.1/5), A.18.5(14/5), A.18.5(31/2)
in Ada.Containers.Hashed_Sets A.18.8(15.1/5), A.18.8(15/5), A.18.8(52/2)
in Ada.Containers.Indefinite_Holders A.18.18(12/5)
in Ada.Containers.Multiway_Trees A.18.10(24.1/5), A.18.10(24/5)
in Ada.Containers.Ordered_Maps A.18.6(13.1/5), A.18.6(13/5), A.18.6(39/2)
in Ada.Containers.Ordered_Sets A.18.9(14.1/5), A.18.9(14/5), A.18.9(65/2)
in Ada.Containers.Vectors A.18.2(27/5), A.18.2(28.1/5), A.18.2(28/5)
in Ada.Strings.Bounded A.4.4(26)
in Ada.Strings.Unbounded A.4.5(20)
element type
container aggregate 4.3.5(7/5), 4.3.5(9/5)
Element_Count
in Ada.Streams.Storage 13.13.1(14/5)
in Ada.Streams.Storage.Bounded 13.13.1(30/5)
in Ada.Streams.Storage.Unbounded 13.13.1(22/5)
elementary type 1.3.1(30/5), 3.2(2/5)
Elementary_Functions
child of Ada.Numerics A.5.1(9/1)
eligible
a type, for a convention B.1(14/3)
else part
of a selective_accept 9.7.1(11)
EM
in Ada.Characters.Latin_1 A.3.3(6)
embedded systems C(1), D(1/5)
Empty
in Ada.Containers.Doubly_Linked_Lists A.18.3(10.3/5)
in Ada.Containers.Hashed_Maps A.18.5(7.3/5)
in Ada.Containers.Hashed_Sets A.18.8(8.2/5)
in Ada.Containers.Indefinite_Holders A.18.18(8.2/5)
in Ada.Containers.Multiway_Trees A.18.10(15.3/5)
in Ada.Containers.Ordered_Maps A.18.6(8.3/5)
in Ada.Containers.Ordered_Sets A.18.9(9.2/5)
in Ada.Containers.Vectors A.18.2(12.4/5)
empty element
of a vector A.18.2(4/2)
empty holder A.18.18(3/3)
Empty_Holder
in Ada.Containers.Indefinite_Holders A.18.18(7/3)
Empty_List
in Ada.Containers.Doubly_Linked_Lists A.18.3(8/2), A.18.3(51.4/5)
Empty_Map
in Ada.Containers.Hashed_Maps A.18.5(5/2), A.18.5(37.5/5)
in Ada.Containers.Ordered_Maps A.18.6(6/2), A.18.6(51.6/5)
Empty_Set
in Ada.Containers.Hashed_Sets A.18.8(5/2), A.18.8(59.4/5)
in Ada.Containers.Ordered_Sets A.18.9(6/2), A.18.9(74.4/5)
Empty_Tree
in Ada.Containers.Multiway_Trees A.18.10(10/3), A.18.10(70.4/5)
Empty_Vector
in Ada.Containers.Vectors A.18.2(10/2), A.18.2(79.4/5)
enabled
invariant expression 7.3.2(21/4)
postcondition expression 6.1.1(19/5)
precondition expression 6.1.1(19/5)
predicate checks 3.2.4(7/3)
encapsulation
See package 7(1)
enclosing
immediately 8.1(13)
Encode
in Ada.Strings.UTF_Encoding.Strings A.4.11(23/3), A.4.11(24/3), A.4.11(25/3)
in Ada.Strings.UTF_Encoding.Wide_Strings A.4.11(31/3), A.4.11(32/3), A.4.11(33/3)
in Ada.Strings.UTF_Encoding.Wide_Wide_Strings A.4.11(39/3), A.4.11(40/3), A.4.11(41/3)
Encoding
in Ada.Strings.UTF_Encoding A.4.11(13/3)
encoding scheme A.4.11(46/3)
Encoding_Error
in Ada.Strings.UTF_Encoding A.4.11(8/3)
Encoding_Scheme
in Ada.Strings.UTF_Encoding A.4.11(4/3)
end of a line 2.2(2/3)
End_Error
in Ada.Direct_IO A.8.4(18)
in Ada.IO_Exceptions A.13(4)
in Ada.Sequential_IO A.8.1(15)
in Ada.Streams.Stream_IO A.12.1(26)
in Ada.Text_IO A.10.1(85/5)
End_Of_File
in Ada.Direct_IO A.8.4(16)
in Ada.Sequential_IO A.8.1(13)
in Ada.Streams.Stream_IO A.12.1(12)
in Ada.Text_IO A.10.1(34/5)
End_Of_Line
in Ada.Text_IO A.10.1(30/5)

End_Of_Page
  in Ada.Text_IO A.10.1(33/5)
End_Search
  in Ada.Directories A.16(33/2)
endian
  big 13.5.3(2)
  little 13.5.3(2)
ENQ
  in Ada.Characters.Latin_1 A.3.3(5)
Enqueue
  in Ada.Containers.Bounded_Priority_Queues A.18.31(5/3)
  in Ada.Containers.Bounded_Synchronized_Queues A.18.29(5/3)
  in Ada.Containers.Synchronized_Queue_Interfaces A.18.27(5/5)
  in Ada.Containers.Unbounded_Priority_Queues A.18.30(5/3)
  in Ada.Containers.Unbounded_Synchronized_Queues A.18.28(5/3)
entity 3.1(12.b)
  [*partial*] 3.1(1/5)
entity with runtime name text C.5(1.1/4)
entry
  closed 9.5.3(5)
  open 9.5.3(5)
  single 9.5.2(20)
entry call 9.5.3(1)
  simple 9.5.3(1)
entry calling convention 6.3.1(13/4)
entry family 9.5.2(20)
entry index subtype 3.8(18/2), 9.5.2(20)
entry queue 9.5.3(12)
entry queuing policy 9.5.3(17)
  default policy 9.5.3(17)
entry_barrier 9.5.2(7)
  used 9.5.2(5/5), P.1
entry_body 9.5.2(5/5)
  used 9.4(8/4), P.1
entry_body_formal_part 9.5.2(6)
  used 9.5.2(5/5), P.1
entry_call_alternative 9.7.2(3/2)
  used 9.7.2(2), 9.7.3(2), P.1
entry_call_statement 9.5.3(2)
  used 5.1(4/2), 9.7.2(3.1/2), P.1
entry_declaration 9.5.2(2/3)
  used 9.1(5/1), 9.4(5/1), P.1
entry_index 9.5.2(4)
  used 9.5.2(3), P.1
entry_index_specification 9.5.2(8/5)
  used 9.5.2(6), P.1
Enum_Rep attribute 13.4(10.2/5)
Enum_Val attribute 13.4(10.5/5)
enumeration literal 3.5.1(6/3)
enumeration type 1.3.1(31/5), 3.2(3), 3.5.1(1)
enumeration_aggregate 13.4(3)
  used 13.4(2), P.1
Enumeration_IO
  in Ada.Text_IO A.10.1(79)
enumeration_literal_specification 3.5.1(3)
  used 3.5.1(2), P.1
enumeration_representation_clause 13.4(2)
  used 13.1(2/1), P.1
enumeration_type_definition 3.5.1(2)
  used 3.2.1(4/2), P.1
environment 10.1.4(1)
environment declarative_part 10.1.4(1)
  for the environment task of a partition 10.2(13)
environment task 10.2(8)
environment variable A.17(1/2)
Environment_Task
  in Ada.Task_Identification C.7.1(3/5)
Environment_Variables
  child of Ada A.17(3/5)
EOT
  in Ada.Characters.Latin_1 A.3.3(5)
EPA
  in Ada.Characters.Latin_1 A.3.3(18)
epoch D.8(19)
equal operator 4.5(1), 4.5.2(1)
Equal_Case_Insensitive
  child of Ada.Strings A.4.10(2/5)
  child of Ada.Strings.Bounded A.4.10(7/5)
  child of Ada.Strings.Fixed A.4.10(5/3)
  child of Ada.Strings.Unbounded A.4.10(10/5)
Equal_Element
  in Ada.Containers.Indefinite_Holders A.18.18(7.1/5)
  in Ada.Containers.Multiway_Trees A.18.10(11.1/5)
Equal_Subtree
  in Ada.Containers.Multiway_Trees A.18.10(14/3)
equality operator 4.5.2(1)
  special inheritance rule for tagged types 3.4(17/2), 4.5.2(14/3)
equals sign 2.1(15/5)
Equals_Sign
  in Ada.Characters.Latin_1 A.3.3(10)
equivalence of use_clauses and selected_components 8.4(1.a)
equivalent element
  of a hashed set A.18.8(64/2)
  of an ordered set A.18.9(78/2)
equivalent key
  of a hashed map A.18.5(42/2)
  of an ordered map A.18.6(55/2)
Equivalent_Elements
  in Ada.Containers.Hashed_Sets A.18.8(46/5), A.18.8(47/5), A.18.8(48/5)
  in Ada.Containers.Ordered_Sets A.18.9(3/2)
Equivalent_Keys
  in Ada.Containers.Hashed_Maps A.18.5(34/5), A.18.5(35/5), A.18.5(36/5)
  in Ada.Containers.Ordered_Maps A.18.6(3/5)
  in Ada.Containers.Ordered_Sets A.18.9(63/2)
Equivalent_Sets
  in Ada.Containers.Hashed_Sets A.18.8(8/2)
  in Ada.Containers.Ordered_Sets A.18.9(9/2)
erroneous execution 1.1.2(32), 1.1.5(10/5)
  cause 3.7.2(4), 3.9(25.3/2), 6.4.1(18/3), 9.8(21), 9.10(19/5), 11.5(26/5), 13.3(13/3), 13.3(27), 13.3(28/2), 13.9.1(8), 13.9.1(12/3), 13.9.1(13/3), 13.11(21), 13.11.2(16/3), 13.11.4(31.1/4), 13.13.2(53/2), A.10.3(22/1), A.12.1(36.1/1), A.13(17), A.17(28/2), A.18.2(252/2), A.18.3(157/2), A.18.4(80/2), A.18.7(101/2), A.18.18(70/3), A.18.19(11/3), A.18.20(15/3), A.18.21(16/3), A.18.22(13/3), A.18.23(16/3), A.18.24(13/3), A.18.25(15/3), B.1(38.1/5), B.3.1(51), B.3.1(55), B.3.1(56), B.3.1(57), B.3.2(35), B.3.2(36), B.3.2(37), B.3.2(38), B.3.2(39), B.3.2(42), C.3.1(14), C.3.1(14.1/3), C.7.1(18), C.7.2(14), C.7.2(15), C.7.2(15.1/2), D.2.6(31/2), D.5.1(12), D.11(9), D.14(19/2), D.14.1(25/2), D.14.2(35/2), H.4(26), H.4(27)
error
  compile-time 1.1.2(27), 1.1.5(4)
  link-time 1.1.2(29), 1.1.5(4)
  run-time 1.1.2(30), 1.1.5(6), 11.5(2/3), 11.6(1/3)
  See also bounded error, erroneous execution
ESA
  in Ada.Characters.Latin_1 A.3.3(17)
ESC
  in Ada.Characters.Latin_1 A.3.3(6)
Establish_RPC_Receiver
  in System.RPC E.5(12)
ETB
  in Ada.Characters.Latin_1 A.3.3(6)
ETX
  in Ada.Characters.Latin_1 A.3.3(5)
evaluable 3.1(11.f)
evaluated operative constituent 4.4(9.2/5)
evaluation 1.3.4(5/5), 3.1(11)
  aggregate 4.3(5)
  allocator 4.8(7/2)
  array_aggregate 4.3.3(21/5)
  attribute_reference 4.1.4(11/5)
  case_expression 4.5.7(21/3)
  concatenation 4.5.3(5)
  dereference 4.1(13)

discrete_range 3.6.1(8)
extension_aggregate 4.3.2(7)
generalized_reference 4.1.5(8/3)
generic_association 12.3(21)
generic_association for a formal object of mode in 12.4(11)
if_expression 4.5.7(20/3)
indexed_component 4.1.1(7)
initialized allocator 4.8(7/2)
membership test 4.5.2(27/4)
name 4.1(11/2)
name that has a prefix 4.1(12)
null literal 4.2(9/5)
numeric literal 4.2(9/5)
parameter_association 6.4.1(7)
prefix 4.1(12)
primary that is a name 4.4(10)
qualified_expression 4.7(4/5)
quantified_expression 4.5.8(6/5)
range 3.5(9)
range_attribute_reference 4.1.4(11/5)
record_aggregate 4.3.1(18)
record_component_association_list 4.3.1(19/5)
reduction_attribute_reference 4.5.10(24/5)
selected_component 4.1.3(14)
short-circuit control form 4.5.1(7)
slice 4.1.2(7)
string_literal 4.2(10/5)
uninitialized allocator 4.8(8)
Val 3.5.5(7), K.2(261)
Value 3.5(55/5)
value conversion 4.6(28)
value_sequence 4.5.10(20/5)
view conversion 4.6(52)
Wide_Value 3.5(43/5)
Wide_Wide_Value 3.5(39.4/3)
exception 1.3.5(2/5), 11(1/3), 11.1(1)
exception occurrence 1.3.5(3/5), 11(1/3)
exception_choice 11.2(5)
used 11.2(3/5), P.1
exception_declaration 11.1(2/3)
used 3.1(3/3), P.1
exception_handler 11.2(3/5)
used 11.2(2), P.1
Exception_Id
in Ada.Exceptions 11.4.1(2/5)
Exception_Identity
in Ada.Exceptions 11.4.1(5/2)
Exception_Information
in Ada.Exceptions 11.4.1(5/2)
Exception_Message
in Ada.Exceptions 11.4.1(4/3)
Exception_Name
in Ada.Exceptions 11.4.1(2/5), 11.4.1(5/2)
Exception_Occurrence
in Ada.Exceptions 11.4.1(3/5)
Exception_Occurrence_Access
in Ada.Exceptions 11.4.1(3/5)
exception_renaming_declaration 8.5.2(2/3)
used 8.5.2(2), P.1
Exceptions
child of Ada 11.4.1(2/5)
Exchange
child of System.Atomic_Operations C.6.2(5/5)
Exchange_Handler
in Ada.Interrupts C.3.2(8)
Exclamation
in Ada.Characters.Latin_1 A.3.3(8)
exclamation point 2.1(15/5)
Exclude
in Ada.Containers.Hashed_Maps A.18.5(24/5)
in Ada.Containers.Hashed_Sets A.18.8(23/5), A.18.8(54/5)
in Ada.Containers.Ordered_Maps A.18.6(23/5)
in Ada.Containers.Ordered_Sets A.18.9(22/5), A.18.9(67/5)
excluded from stable property checks 7.3.4(24/5)
excludes null
subtype 3.10(13.1/2)
exclusive
protected operation 9.5.1(2.4/4)
Exclusive_Functions aspect 9.5.1(2.2/4)
executable 3.1(11.f)
execution 1.3.4(7/5), 3.1(11)
abort_statement 9.8(4)
aborting the execution of a construct 9.8(5)
accept_statement 9.5.2(24/5)
Ada program 9(1/5)
assignment_statement 5.2(7), 7.6(17), 7.6.1(12/2)
asynchronous_select with a delay_statement trigger 9.7.4(7)
asynchronous_select with a procedure call trigger 9.7.4(6/2)
asynchronous_select with an entry call trigger 9.7.4(6/2)
block_statement 5.6(5)
call on a dispatching operation 3.9.2(14)
call on an inherited subprogram 3.4(27/2)
case_statement 5.4(11/3)
conditional_entry_call 9.7.3(3)
delay_statement 9.6(20)
dynamically enclosing 11.4(2)
entry_body 9.5.2(26)
entry_call_statement 9.5.3(8)
exit_statement 5.7(5)
extended_return_statement 6.5(5.12/5)
goto_statement 5.8(5)
handled_sequence_of_statements 11.2(10)
handler 11.4(7)
if_statement 5.3(5/3)
included by another execution 11.4(2.a)
instance of Unchecked_Deallocation 7.6.1(10)
loop_statement 5.5(7/5)
loop_statement with a for loop_parameter_specification 5.5(9/5)
loop_statement with a while iteration_scheme 5.5(8)
null_statement 5.1(13)
partition 10.2(25)
pragma 2.8(12)
program 10.2(25)
protected subprogram call 9.5.1(3)
raise_statement with an exception_name 11.3(4/4)
re-raise statement 11.3(4/4)
remote subprogram call E.4(9)
requeue protected entry 9.5.4(9)
requeue task entry 9.5.4(8)
requeue_statement 9.5.4(7/5)
selective_accept 9.7.1(15)
sequence_of_statements 5.1(15)
simple_return_statement 6.5(6/2)
subprogram call 6.4(10/2)
subprogram_body 6.3(7)
task 9.2(1)
task_body 9.2(1)
timed_entry_call 9.7.2(4/2)
execution resource
associated with a protected object 9.4(18)
required for a task to run 9(10/5)
execution time
of a task D.14(11/3)
Execution_Time
child of Ada D.14(3/5)
exhaust
a budget D.14.2(14/2)
exist
cease to 7.6.1(11/3), 13.11.2(10/4), 13.11.5(7.1/4)
Exists
in Ada.Directories A.16(24/2)
in Ada.Environment_Variables A.17(5/2)
exit_statement 5.7(2)
used 5.1(4/2), P.1
Exit_Status
in Ada.Command_Line A.15(7)
Exp
in Ada.Numerics.Generic_Complex_Elementary_Functions G.1.2(3)
in Ada.Numerics.Generic_Elementary_Functions A.5.1(4)
expanded name 4.1.3(4)
Expanded_Name
in Ada.Tags 3.9(7/2)
expected profile 8.6(26)
accept_statement entry_direct_name 9.5.2(11)
Access attribute_reference prefix 3.10.2(2.3/2)
attribute_definition_clause name 13.3(4)
character_literal 4.2(3)
formal subprogram actual 12.6(6)

formal subprogram default_name 12.6(5)
name in an aspect_specification 13.1.1(8/3)
subprogram_renaming_declaration 8.5.4(3)
expected type 8.6(20/2)
abort_statement task_name 9.8(3/5)
access attribute_reference 3.10.2(2/2)
Access attribute_reference prefix 3.10.2(2.3/2)
actual parameter 6.4.1(3)
aggregate 4.3(3/5)
allocator 4.8(3/3)
array delta aggregate 4.3.4(6/5)
array_aggregate 4.3.3(7/2)
array_aggregate component expression 4.3.3(7/2)
array_aggregate discrete_choice 4.3.3(8)
assignment_statement expression 5.2(4/2)
assignment_statement variable_name 5.2(4/2)
Attach_Handler pragma second argument J.15.7(6/3)
attribute_definition_clause expression or name 13.3(4)
attribute_designator expression 4.1.4(7)
base expression of a delta aggregate 4.3.4(7/5)
case_expression selecting_expression 4.5.7(15/3)
case_expression_alternative discrete_choice 4.5.7(15/3)
case_statement selecting_expression 5.4(4/3)
case_statement_alternative discrete_choice 5.4(4/3)
character_literal 4.2(3)
code_statement 13.8(4)
component_clause expressions 13.5.1(7)
component_declaration default_expression 3.8(7)
condition 4.5.7(14/3)
CPU pragma argument J.15.9(3/3)
decimal fixed point type digits 3.5.9(6)
delay_relative_statement expression 9.6(5)
delay_until_statement expression 9.6(5)
delta aggregate base expression 4.3.4(7/5)
delta_constraint expression J.3(3/4)
dependent_expression 4.5.7(8/3)
dereference name 4.1(8)
discrete_subtype_definition range 3.6(8)
discriminant default_expression 3.7(7)
discriminant_association expression 3.7.1(6)
Dispatching_Domains pragma argument J.15.10(3/3)
entry_index 9.5.2(11)
enumeration_representation_clause expressions 13.4(4)
expression in an aspect_specification 13.1.1(7/3)
expression of a Default_Component_Value aspect 3.6(22.4/3)
expression of a Default_Value aspect 3.5(56.5/3)
expression of a predicate aspect 3.2.4(2/3)
expression of expression function 6.8(3/4)
expression of extended_return_object_declaration 6.5(3/5)
expression of simple_return_statement 6.5(3/5)
extension_aggregate 4.3.2(4/2)
extension_aggregate ancestor expression 4.3.2(4/2)
external name J.15.5(6/3)
first_bit 13.5.1(7)
fixed point type delta 3.5.9(6)
generic formal in object actual 12.4(4)
generic formal object default_expression 12.4(3)
index_constraint discrete_range 3.6.1(4)
indexable_container_object_prefix 4.1.6(11/3)
indexed_component expression 4.1.1(4)
integer_literal 4.2(4/5)
Interrupt_Priority pragma argument J.15.11(5/3)
invariant expression 7.3.2(4/3)
iterable_name 5.5.2(3/3)
iterator_name 5.5.2(3/3)
last_bit 13.5.1(7)
link name J.15.5(6/3)
linker options B.1(10.1/3)
membership test simple_expression 4.5.2(3/3)
modular_type_definition expression 3.5.4(5/5)
name in an aspect_specification 13.1.1(7/3)
number_declaration expression 3.3.2(3)
object in an aspect_specification 13.1.1(6/3)
object_declaration initialization expression 3.3.1(4)
parameter default_expression 6.1(17)
position 13.5.1(7)
postcondition expression 6.1.1(6/3)
precondition expression 6.1.1(6/3)
Priority pragma argument J.15.11(5/3)
quantified_expression 4.5.8(5/3)
range simple_expressions 3.5(5)
range_attribute_designator expression 4.1.4(7)
range_constraint range 3.5(5)
real_literal 4.2(4/5)
real_range_specification bounds 3.5.7(5/5)
record delta aggregate 4.3.4(5/5)
record_aggregate 4.3.1(8/2)
record_component_association expression 4.3.1(10)
reference_object_name 4.1.5(5/3)
Relative_Deadline pragma argument J.15.12(3/3)
requested decimal precision 3.5.7(4)
restriction parameter expression 13.12(5)
selecting_expression case_expression 4.5.7(15/3)
selecting_expression case_statement 5.4(4/3)
short-circuit control form relation 4.5.1(1)
signed_integer_type_definition simple_expression 3.5.4(5/5)
slice discrete_range 4.1.2(4)
Storage_Size pragma argument J.15.4(4/3)
string_literal 4.2(4/5)
subpool_handle_name 4.8(3/3)
type_conversion operand 4.6(6)
Unchecked_Access attribute 13.10(4.a)
variant_part discrete_choice 3.8.1(6)
expiration time [*partial*] 9.6(1)
for a delay_relative_statement 9.6(20)
for a delay_until_statement 9.6(20)
expires
execution timer D.14.1(15/3)
explicit class-wide postcondition expression 7.3.4(23/5)
explicit declaration 3.1(5)
explicit initial value 3.3.1(1/3)
explicit specific postcondition expression 7.3.4(23/5)
explicit_actual_parameter 6.4(6)
used 6.4(5), P.1
explicit_dereference 4.1(5)
used 4.1(2/5), P.1
explicit_generic_actual_parameter 12.3(5)
used 12.3(4), P.1
explicitly aliased parameter 6.1(23.1/3)
explicitly assign 10.2(2/5)
explicitly limited record 3.8(13.1/3)
exponent 2.4.1(4), 4.5.6(11/3)
used 2.4.1(2), 2.4.2(2), P.1
Exponent attribute A.5.3(18)
exponentiation operator 4.5(1), 4.5.6(7)
Export aspect B.1(1/3)
Export pragma J.15.5(3/3), L(13.1/3)
exported entity B.1(23/3)
expression 4.4(1/3), 4.4(2)
case 4.5.7(1/3)
conditional 4.5.7(1/3)
declare 4.5.9(1/5)
if 4.5.7(1/3)
of a return statement 6.5(5.b.2/5)

of an extended_return_statement 6.5(5.1/5)
parenthesized 4.4(9.1/5)
predicate-static 3.2.4(15/3)
quantified 4.5.8(0.1/4)
reduction 4.5.10(1/5)
used 2.8(3/3), 3.3.1(2/3), 3.3.2(2), 3.5.4(4), 3.5.7(2), 3.5.9(3), 3.5.9(4), 3.7(6), 3.7.1(3/5), 4.1.1(2), 4.1.4(3/2), 4.1.4(5), 4.3.1(4/2), 4.3.2(3), 4.3.3(3/5), 4.3.3(5.1/5), 4.3.3(5/5), 4.3.4(3/5), 4.3.4(4/5), 4.3.5(15/5), 4.3.5(18/5), 4.3.5(20/5), 4.3.5(21/5), 4.4(7/5), 4.5.7(3/3), 4.5.7(4/3), 4.5.7(5/3), 4.5.7(6/3), 4.5.8(3/3), 4.5.9(2/5), 4.5.10(5/5), 4.6(2), 4.7(2), 5.2(2), 5.4(2/3), 6.4(6), 6.5(2.1/5), 6.5(2/2), 6.8(2/4), 9.5.2(4), 9.6(3), 9.6(4), 11.3(2/2), 11.4.2(3/2), 12.3(5), 13.1.1(4/5), 13.3(2), 13.5.1(4), 13.12(4.1/2), B.1(8), B.1(10.1/3), D.2.2(2.2/2), J.7(1), J.8(1), J.15.4(2/3), J.15.5(2/3), J.15.5(3/3), J.15.7(4/3), J.15.9(2/3), L(2.2/2), L(6.1/3), L(8.2/3), L(13.1/3), L(14.1/3), L(19), L(27.2/2), L(35.1/3), P.1
expression function 6.8(3.1/5)
static 6.8(3.6/5)
expression_function_declaration 6.8(2/4)
used 3.1(3/3), 9.4(8/4), P.1
extended_digit 2.4.2(5)
used 2.4.2(4), P.1
extended_global_mode H.7(3/5)
used 6.1.2(4/5), P.1
Extended_Index *subtype of* Index_Type'Base
in Ada.Containers.Vectors A.18.2(7/2)
extended_return_object_declaration 6.5(2.1/5)
used 6.5(2.2/3), P.1
extended_return_statement 6.5(2.2/3)
used 5.1(5/5), P.1
extension
of a private type 3.9(2.1/2), 3.9.1(1/2)
of a record type 3.9(2.1/2), 3.9.1(1/2)
of a type 3.9(2/2), 3.9.1(1/2)
in Ada.Directories A.16(18/5)
extension_aggregate 4.3.2(2)
used 4.3(2/5), P.1
extensions to Ada 2005 1.1.2(39.ff/3), 2.2(14.d/3), 2.8(19.f/3), 3.2.1(16.g/3), 3.2.2(15.e/3), 3.2.4(52.b/3), 3.3(26.j/3), 3.3.1(33.l/3), 3.5(63.n/3), 3.5.5(17.b/3), 3.6(30.j/3), 3.7(37.m/3), 3.8(31.j/3), 3.8.1(29.f/3), 3.9.3(17.l/3), 3.10.1(23.m/3), 3.10.2(41.r/3), 4.1(17.h/3), 4.1.5(16.a/3), 4.1.6(23.a/3), 4.3.1(31.f/3), 4.5.2(39.l/3), 4.5.7(25.a/3), 4.5.8(13.a/3), 4.8(20.q/3), 5.1(22.e/3), 5.5.1(22.b/3), 5.5.2(17.a/3), 6.1(42.m/3), 6.1.1(44.a/3), 6.3(11.g/3), 6.3.2(7.d/3), 6.5(28.s/3), 6.5.1(11.b/3), 6.7(7.b/3), 6.8(10.a/3), 7.1(17.e/3), 7.2(15.f/3), 7.3(25.g/3), 7.3.2(41.a/3), 7.6(27.l/3), 8.4(16.h/3), 8.5.1(10.h/3), 8.5.2(6.a/3), 8.5.3(6.b/3), 8.5.4(21.g/3), 8.5.5(7.c/3), 8.6(34.w/3), 9.1(32.j/3), 9.4(35.h/3), 9.5(72.b/3), 9.5.2(37.e/3), 9.5.4(20.b/3), 10.1.3(24.d/3), 10.2.1(28.m/3), 11.1(8.c/3), 11.4.2(28.c/3), 12.1(24.d/3), 12.3(29.j/3), 12.4(12.i/3), 12.5(16.g/3), 12.5.1(28.j/3), 12.6(24.e/3), 12.7(25.f/3), 13.1.1(39.b/3), 13.2(9.d/3), 13.3(86.o/3), 13.11.1(5.d/3), 13.11.3(9.c/4), 13.11.4(35.a/3), 13.11.5(10.f/3), 13.12.1(13.c/3), 13.13.2(60.v/3), A.3.5(64.a/3), A.3.6(1.a/3), A.4.7(49.g/3), A.4.8(51.b/3), A.4.9(12.d/3), A.4.10(22.a/3), A.4.11(108.a/3), A.12.1(36.g/3), A.16.1(38.b/3), A.18.2(264.d/3), A.18.3(164.d/3), A.18.4(84.d/3), A.18.5(62.e/3), A.18.6(95.f/3), A.18.7(105.d/3), A.18.8(88.e/3), A.18.9(116.e/3), A.18.10(233.c/3), A.18.17(12.a/3), A.18.18(74.b/3), A.18.19(17.a/3), A.18.20(20.a/3), A.18.21(22.a/3), A.18.22(19.a/3), A.18.23(21.a/3), A.18.24(18.a/3), A.18.25(20.a/3), A.18.26(13.b/3), A.18.27(17.b/3), A.18.28(12.c/3), A.18.29(13.a/3), A.18.30(17.c/3), A.18.31(14.a/3), A.19(12.b/3), B.1(51.e/3), B.3.3(32.c/3), C.3.1(25.d/3), C.6(26.f/3), D.1(29.g/3), D.2.4(11.b/3), D.2.6(34.c/3), D.7(22.j/3), D.10(12.b/3), D.10.1(15.a/3), D.14.3(7.a/3), D.16(16.c/3), D.16.1(34.b/3), E.2.1(11.e/5), E.2.2(20.k/3), E.2.2(20.m/5), E.2.3(20.f/3), E.4.1(10.b/3), H.4(28.h/3), J.15.9(6.a/3), J.15.10(5.a/3)
extensions to Ada 2012 1.1.2(39.ss/5), 3.2.4(52.d/4), 3.3(26.p/5), 3.3.2(10.d/5), 3.5(63.o/4), 3.7(37.o/5), 3.8(31.k/5), 3.10(26.o/5), 3.10.2(41.jj/5), 4.2.1(24.a/5), 4.3.1(31.h/5), 4.3.3(49.l/5), 4.3.4(30.a/5), 4.3.5(91.a/5), 4.5.9(12.a/5), 4.5.10(52.a/5), 4.9(44.bb/5), 4.10(56.b/5), 5.1(22.f/5), 5.2.1(8.a/5), 5.5(32.c/5), 5.5.1(22.e/5), 5.5.2(17.e/5), 5.5.3(37.a/5), 5.6.1(9.a/5), 6.1(42.q/5), 6.1.1(44.h/5), 6.1.2(44.a/5), 6.3.1(26.m/4), 6.5(28.bb/5), 6.5.1(11.d/5), 6.8(10.b/4), 7.3.2(41.i/4), 7.3.3(12.a/5), 7.3.4(32.a/5), 7.5(26.k/5), 8.1(18.r/4), 8.5.1(10.k/5), 9.4(35.k/5), 9.5(72.f/5), 9.5.1(25.b/4), 9.5.2(37.f/5), 9.10.1(26.b/5), 10.2.1(28.u/5), 11.3(8.c/4), 11.5(31.m/5), 12.4(12.j/5), 12.5(16.k/5), 12.6(24.h/5), 13.1(29.x/5), 13.1.1(39.f/5), 13.3(86.s/5), 13.4(16.e/5), 13.5.1(32.f/5), 13.11(43.m/5), 13.11.3(9.e/4), 13.12.1(13.e/5), 13.13.1(39.e/5), 13.13.2(60.dd/4), A.4.12(36.a/5), A.5.2(62.d/5), A.5.5(3.a/5), A.5.6(25.b/5), A.5.7(28.b/5), A.12.1(36.i/4), A.15.1(3.a/5), A.16.2(3.a/5), A.17.1(3.a/5), A.18(12.f/5), A.18.2(264.q/5), A.18.3(164.o/5), A.18.4(84.l/5), A.18.5(62.h/5), A.18.6(95.j/5), A.18.7(105.l/5), A.18.8(88.i/5), A.18.9(116.j/5), A.18.10(233.f/5), A.18.32(16.b/5), B.1(51.g/4), B.3(84.k/4), C.6(26.k/5), C.6.1(6.a/5), C.6.2(24.a/5), C.6.3(11.a/5), C.6.4(18.a/5), C.6.5(18.a/5), D.2.1(17.e/5), D.3(23.f/5), D.4(23.f/5), D.4.1(8.b/5), D.5.2(11.b/5), D.7(22.m/4), D.13(13.e/5), D.16(16.d/5), H.4(28.m/5), H.4.1(5.c/5), H.5(7.b/5), H.7(9.a/5), H.7.1(20.a/5)

extensions to Ada 83 1.1.2(39.g), 2.1(19.b), 2.8(19.d), 2.8(29.a), 3.2.3(8.b), 3.3(26.a), 3.3.1(33.a), 3.3.2(10.a), 3.4(38.d), 3.5(63.b), 3.5.2(11.f), 3.5.4(36.a), 3.5.5(17.a), 3.5.9(28.b/3), 3.6(30.a), 3.6.1(18.a), 3.6.3(8.e), 3.7(37.a), 3.7.2(4.b/1), 3.8(31.a), 3.8.1(29.a), 3.9(33.a), 3.9.1(17.a), 3.9.2(24.a), 3.10(26.a), 3.10.1(23.a), 3.10.2(41.a), 3.11(14.a/2), 4.1(17.a), 4.1.3(19.a), 4.1.4(19.a), 4.2(14.b), 4.3(6.b), 4.3.1(31.a), 4.3.2(13.a), 4.3.3(49.a), 4.4(15.a), 4.5.2(39.a), 4.5.3(14.d), 4.5.5(35.a), 4.6(71.d), 4.8(20.b), 4.9(44.a), 5.1(22.a), 5.2(29.a), 5.4(18.a), 6.1(42.a), 6.2(15.a), 6.3(11.a), 6.3.1(26.a), 6.4.1(19.b), 6.6(9.a), 7.3(25.a), 7.4(14.a), 7.5(26.a), 7.6(27.b), 8.2(13.b), 8.3(29.p), 8.4(16.e), 8.5.5(7.a), 8.6(34.b), 9.1(32.a/1), 9.4(35.a/3), 9.5.2(37.a), 9.5.4(20.a), 9.6(40.b), 9.7(4.a), 9.7.4(14.a), 10.1.1(35.n), 10.1.2(32.a), 10.1.3(24.a), 10.1.4(10.b/2), 10.2(34.d), 10.2.1(28.c), 11.2(12.a), 11.4.1(20.a), 11.5(31.a), 12.1(24.a), 12.3(29.c), 12.4(12.b), 12.5.4(13.b), 12.7(25.a), 13.1(29.b/1), 13.3(86.a), 13.4(16.a), 13.5.3(9.a), 13.7(38.a.1/1), 13.8(14.a), 13.9.2(14.d), 13.11(43.a), 13.12(17.b), 13.13(1.c), 13.14(20.e), A.1(56.d), A.2(4.e/3), A.3(1.a/3), A.4(1.a/3), A.5(5.b), A.5.3(72.g), A.5.4(4.c), A.6(1.b), A.9(11.a/3), A.10(11.a), A.10.1(86.b), A.11(5.a), A.15(22.b/3), B(2.b), B.1(51.a), C(1.a), D(6.a), D.1(29.b), E(1.a), F(7.b), G(7.a), G.2(3.a), G.2.1(16.g), H(6.b), J.7(2.c), J.15.1(6.b/3)

extensions to Ada 95 1.1.2(39.s/2), 2.1(19.f/2), 2.3(8.c/2), 3.2.2(15.d/2), 3.3.1(33.g/2), 3.3.1(33.h/2), 3.4(38.i/2), 3.5(63.j/2), 3.5.2(11.l/2), 3.5.4(36.k/2), 3.6(30.g/2), 3.6.3(8.h/2), 3.8(31.e/2), 3.8(31.f/2), 3.9(33.e/2), 3.9.1(17.b/2), 3.9.2(24.c/2), 3.9.3(17.d/2), 3.9.4(36.a/2), 3.10(26.g/2), 3.10.1(23.j/2), 3.10.2(41.g/2), 4.1.3(19.e/2), 4.2(14.d/2), 4.3(6.k/2), 4.3.1(31.c/2), 4.3.3(49.f/2), 4.5.2(39.f/2), 4.6(71.l/2), 4.8(20.h/2), 5.1(22.d/2), 6.1(42.f/2), 6.4(31.d/2), 6.5(28.i/2), 6.5.1(11.a/2), 6.7(7.a/2), 7.3(25.d/2), 7.4(14.j/2), 7.5(26.c/2), 7.6(27.c/2), 8.3.1(16.b/2), 8.5.1(10.d/2), 8.5.4(21.a/2), 8.6(34.q/2), 9.1(32.e/2), 9.4(35.b/2), 9.6.1(91.b/2), 9.7.4(14.b/2), 10.1.1(35.r/2), 10.1.2(32.g/2), 10.1.2(32.h/3), 10.1.3(24.b/2), 10.2.1(28.g/2), 11.3(8.b/2), 11.4.1(20.f/2), 11.4.2(28.a/2), 11.5(31.g/2), 12.3(29.i/2), 12.4(12.e/2), 12.5.1(28.c/2), 12.5.5(8.a/2), 12.6(24.a/2), 12.7(25.b/2), 13.1(29.i/2), 13.5.1(32.e/2), 13.7(38.e/2), 13.7.1(16.d/2), 13.11(43.e/2), 13.12(17.c/3), 13.12.1(13.a/2), 13.13.1(39.b/2), 13.13.2(60.i/2), A.1(56.h/2), A.3.1(7.b/2), A.3.2(60.c/2), A.3.4(35.a/2), A.4.2(68.a/2), A.4.5(88.d/2), A.4.6(8.b/2), A.4.7(49.b/2), A.4.8(51.a/2), A.4.9(12.c/2), A.5(5.c/2), A.5.3(72.i/2), A.10.7(26.b/2), A.10.11(29.a/2), A.10.12(29.a/2), A.11(5.b/2), A.12.4(5.a/2), A.16(131.c/2), A.17(33.b/2), A.18(12.d/3), A.18.1(8.c/2), A.18.2(264.b/2), A.18.3(164.a/2), A.18.5(62.c/2), A.18.6(95.d/2), A.18.8(88.c/2), A.18.9(116.c/2), A.18.11(13.a/2), A.18.12(12.a/2), A.18.13(13.a/2), A.18.14(13.a/2), A.18.15(5.d/2), A.18.16(5.d/2), A.18.26(13.a/2), B.3(84.b/2), B.3.1(60.b/2), B.3.3(32.a/2), C.7.3(17.a/2), D.2.2(21.a/2), D.2.4(11.a/2), D.2.5(19.a/2), D.2.6(34.b/2), D.3(23.a/2), D.5.1(19.a/2), D.5.2(11.a/2), D.7(22.b/2), D.10(12.a/2), D.11(19.a/2), D.13(13.a/3), D.14(29.b/2), D.14.1(29.a/2), D.14.2(38.a/2), D.15(27.a/2), E.2.2(20.c/3), F.3.5(6.a/2), G.1.1(58.g/2), G.1.3(35.a/2), G.1.5(1.a/2), G.3(1.c/3), G.3.1(91.b/2), G.3.2(161.b/2), H.4(28.c/2), H.5(7.a/2), H.6(17.a/2), J.15.1(6.c/3)

external call 9.5(4/3)

external effect
- of the execution of an Ada program 1.1.3(8)
- volatile/atomic objects C.6(20/5)

external file A.7(1)

external interaction 1.1.3(8)

external name B.1(34)

external requeue 9.5(7)

external streaming
- type supports 13.13.2(52/3)

External_Name aspect B.1(1/3)

External_Tag
- *in* Ada.Tags 3.9(7/2)

External_Tag aspect 13.3(75/3), K.2(65)

External_Tag attribute 13.3(75/3)

External_Tag clause 13.3(7/2), 13.3(75/3), K.2(65)

extra permission to avoid raising exceptions 11.6(5/5)

extra permission to reorder actions 11.6(6/5)

F

factor 4.4(6)
- *used* 4.4(5), P.1

factory 3.9(30/2)

failure
- of a language-defined check 11.5(2/3)
- *in* Ada.Command_Line A.15(8)

fall-back handler C.7.3(9/2)

False 3.5.3(1)

family
- entry 9.5.2(20)

Feminine_Ordinal_Indicator
- *in* Ada.Characters.Latin_1 A.3.3(21/3)

FF
- *in* Ada.Characters.Latin_1 A.3.3(5)

Field *subtype of* Integer
- *in* Ada.Numerics.Big_Numbers.Big_Integers A.5.5(3/5)
- *in* Ada.Text_IO A.10.1(6)

FIFO_Queuing queuing policy D.4(7/5)

FIFO_Within_Priorities task dispatching policy D.2.3(2/2)

file
- as file object A.7(2/3)

file name A.16(46/2)

file terminator A.10(7)

File_Access
- *in* Ada.Text_IO A.10.1(18)

File_Kind
- *in* Ada.Directories A.16(22/2)

File_Mode
- *in* Ada.Direct_IO A.8.4(4)
- *in* Ada.Sequential_IO A.8.1(4)
- *in* Ada.Streams.Stream_IO A.12.1(6)
- *in* Ada.Text_IO A.10.1(4)

File_Size
- *in* Ada.Directories A.16(23/2)

File_Type
- *in* Ada.Direct_IO A.8.4(3)
- *in* Ada.Sequential_IO A.8.1(3)

in Ada.Streams.Stream_IO A.12.1(5/5)
in Ada.Text_IO A.10.1(3)
filter
iterator construct 5.5(6.2/5)
Filter_Type
in Ada.Directories A.16(30/2)
finalization
of a master 7.6.1(4)
of a protected object 9.4(20)
of a protected object C.3.1(12/3)
of a task object J.7.1(8)
of an object 7.6.1(5)
of environment task for a foreign language main subprogram B.1(39/3)
child of Ada 7.6(4/5)
Finalize 7.6(2)
in Ada.Finalization 7.6(6/2), 7.6(8/2)
Find
in Ada.Containers.Doubly_Linked_Lists A.18.3(41/5)
in Ada.Containers.Hashed_Maps A.18.5(30/5)
in Ada.Containers.Hashed_Sets A.18.8(43/5), A.18.8(56/5)
in Ada.Containers.Multiway_Trees A.18.10(38/5)
in Ada.Containers.Ordered_Maps A.18.6(38/5)
in Ada.Containers.Ordered_Sets A.18.9(49/5), A.18.9(69/5)
in Ada.Containers.Vectors A.18.2(68/5)
Find_In_Subtree
in Ada.Containers.Multiway_Trees A.18.10(39.1/5), A.18.10(39/5)
Find_Index
in Ada.Containers.Vectors A.18.2(67/2)
Find_Token
in Ada.Strings.Bounded A.4.4(50.1/3), A.4.4(51)
in Ada.Strings.Fixed A.4.3(15.1/3), A.4.3(16)
in Ada.Strings.Unbounded A.4.5(45.1/3), A.4.5(46)
Fine_Delta
in System 13.7(9)
First
in Ada.Containers.Doubly_Linked_Lists A.18.3(33/5)
in Ada.Containers.Hashed_Maps A.18.5(27/5)
in Ada.Containers.Hashed_Sets A.18.8(40/5)
in Ada.Containers.Ordered_Maps A.18.6(28/5)
in Ada.Containers.Ordered_Sets A.18.9(41/5)
in Ada.Containers.Vectors A.18.2(58/5)
in Ada.Iterator_Interfaces 5.5.1(3/3), 5.5.1(4.6/5)
First attribute 3.5(12), 3.6.2(3)
first element
of a hashed set A.18.8(68/2)
of a set A.18.7(6/2)
of an ordered set A.18.9(81/3)
first node
of a hashed map A.18.5(46/2)
of a map A.18.4(6/2)
of an ordered map A.18.6(58/3)
first subtype 3.2.1(6/5), 3.4.1(5)
First(N) attribute 3.6.2(4)
first_bit 13.5.1(5)
used 13.5.1(3), P.1
First_Bit attribute 13.5.2(3/2)
First_Child
in Ada.Containers.Multiway_Trees A.18.10(60.1/5), A.18.10(60/5)
First_Child_Element
in Ada.Containers.Multiway_Trees A.18.10(61.1/5), A.18.10(61/5)
First_Element
in Ada.Containers.Doubly_Linked_Lists A.18.3(34/5)
in Ada.Containers.Ordered_Maps A.18.6(29/5)
in Ada.Containers.Ordered_Sets A.18.9(42/5)
in Ada.Containers.Vectors A.18.2(59/5)
First_Index
in Ada.Containers.Vectors A.18.2(57/5)
First_Key
in Ada.Containers.Ordered_Maps A.18.6(30/5)
First_Valid attribute 3.5.5(7.2/4)
Fixed
child of Ada.Strings A.4.3(5/5)
fixed point type 3.5.9(1)
Fixed_Conversions
in Ada.Numerics.Big_Numbers.Big_Reals A.5.7(14/5)
Fixed_IO
in Ada.Text_IO A.10.1(68)
fixed_point_definition 3.5.9(2)
used 3.5.6(2), P.1
Float 3.5.7(12), 3.5.7(14)
in Standard A.1(21)
Float_Conversions
in Ada.Numerics.Big_Numbers.Big_Reals A.5.7(13/5)
Float_IO
in Ada.Text_IO A.10.1(63)
Float_Random
child of Ada.Numerics A.5.2(5/5)
Float_Text_IO
child of Ada A.10.9(33)
Float_Wide_Text_IO
child of Ada A.11(2/2)
Float_Wide_Wide_Text_IO
child of Ada A.11(3/2)
Floating
in Interfaces.COBOL B.4(9)
floating point type 3.5.7(1)
floating_point_definition 3.5.7(2)
used 3.5.6(2), P.1
Floor
in Ada.Containers.Ordered_Maps A.18.6(40/5)
in Ada.Containers.Ordered_Sets A.18.9(50/2), A.18.9(70/5)
Floor attribute A.5.3(30)
Flush
in Ada.Direct_IO A.8.4(10.1/5)
in Ada.Sequential_IO A.8.1(10.1/5)
in Ada.Streams.Stream_IO A.12.1(25/1)
in Ada.Text_IO A.10.1(21/5)
Fore attribute 3.5.10(4)
form
of an external file A.7(1)
in Ada.Direct_IO A.8.4(9), A.8.4(18.13/5), A.8.4(18.6/5)
in Ada.Sequential_IO A.8.1(9), A.8.1(15.13/5), A.8.1(15.6/5)
in Ada.Streams.Stream_IO A.12.1(11), A.12.1(26.13/5), A.12.1(26.6/5)
in Ada.Text_IO A.10.1(12), A.10.1(85.13/5), A.10.1(85.6/5)
formal object, generic 12.4(1)
formal package, generic 12.7(1)
formal parameter
of a subprogram 6.1(17)
formal parameter set H.7.1(13/5)
formal subprogram, generic 12.6(1)
formal subtype 12.5(5)
formal type 12.5(5)
formal_abstract_subprogram_declaration 12.6(2.2/3)
used 12.6(2/2), P.1
formal_access_type_definition 12.5.4(2)
used 12.5(3/2), P.1
formal_array_type_definition 12.5.3(2)
used 12.5(3/2), P.1
formal_complete_type_declaration 12.5(2.1/5)
used 12.5(2/3), P.1
formal_concrete_subprogram_declaration 12.6(2.1/3)
used 12.6(2/2), P.1
formal_decimal_fixed_point_definition 12.5.2(7)
used 12.5(3/2), P.1
formal_derived_type_definition 12.5.1(3/5)
used 12.5(3/2), P.1
formal_discrete_type_definition 12.5.2(2)
used 12.5(3/2), P.1
formal_floating_point_definition 12.5.2(5)
used 12.5(3/2), P.1
formal_group_designator H.7.1(5/5)
used H.7.1(4/5), P.1
formal_incomplete_type_declaration 12.5(2.2/5)
used 12.5(2/3), P.1

formal_interface_type_definition 12.5.5(2/2)
used 12.5(3/2), P.1
formal_modular_type_definition 12.5.2(4)
used 12.5(3/2), P.1
formal_object_declaration 12.4(2/3)
used 12.1(6), P.1
formal_ordinary_fixed_point_definition 12.5.2(6)
used 12.5(3/2), P.1
formal_package_actual_part 12.7(3/2)
used 12.7(2/3), P.1
formal_package_association 12.7(3.1/2)
used 12.7(3/2), P.1
formal_package_declaration 12.7(2/3)
used 12.1(6), P.1
formal_parameter_name H.7.1(6/5)
used H.7.1(4/5), P.1
formal_parameter_set H.7.1(4/5)
formal_part 6.1(14)
used 5.5.3(3/5), 6.1(12), 6.1(13/2), P.1
formal_private_type_definition 12.5.1(2)
used 12.5(3/2), P.1
formal_signed_integer_type_definition 12.5.2(3)
used 12.5(3/2), P.1
formal_subprogram_declaration 12.6(2/2)
used 12.1(6), P.1
formal_type_declaration 12.5(2/3)
used 12.1(6), P.1
formal_type_definition 12.5(3/2)
used 12.5(2.1/5), P.1
format_effector 2.1(13/3)
Formatting
child of Ada.Calendar 9.6.1(15/5)
Fortran
child of Interfaces B.5(4/5)
Fortran interface B.5(1/3)
Fortran standard 1.2.1(3/5)
Fortran_Character
in Interfaces.Fortran B.5(12/3)
Fortran_Integer
in Interfaces.Fortran B.5(5)
forward iterator 5.5.2(4/5)
Forward_Iterator
in Ada.Iterator_Interfaces 5.5.1(3/3)
Fraction attribute A.5.3(21)
Fraction_One_Half
in Ada.Characters.Latin_1 A.3.3(22)
Fraction_One_Quarter
in Ada.Characters.Latin_1 A.3.3(22)
Fraction_Three_Quarters
in Ada.Characters.Latin_1 A.3.3(22)
Free
in Ada.Strings.Unbounded A.4.5(7)
in Interfaces.C.Strings B.3.1(11)
freed
See nonexistent 13.11.2(10/4)
freeing storage 13.11.2(1)
freezing
by a constituent of a construct 13.14(4/1)
by an expression 13.14(8/4)
by an implicit call 13.14(8.1/3)
by an object name 13.14(8/4)
class-wide type caused by the freezing of the specific type 13.14(15/5)
constituents of a full type definition 13.14(15/5)
designated subtype caused by an allocator 13.14(13)
entity 13.14(2)
entity caused by a body 13.14(3/5)
entity caused by a construct 13.14(4/1)
entity caused by a name 13.14(11)
entity caused by the end of an enclosing construct 13.14(3/5)
expression of an expression function by a call 13.14(10.1/4)
expression of an expression function by Access attribute 13.14(10.3/4)
expression of an expression function by an instantiation 13.14(10.2/4)
first subtype caused by the freezing of the type 13.14(15/5)
generic_instantiation 13.14(5/3)
nominal subtype caused by a name 13.14(11)
object_declaration 13.14(6)
profile 13.14(2.1/3)
profile of a callable entity by an instantiation 13.14(10.2/4)
profile of a function call 13.14(10.1/4)
specific type caused by the freezing of the class-wide type 13.14(15/5)
subtype caused by a record extension 13.14(7)
subtype caused by an implicit conversion 13.14(8.2/1)
subtype caused by an implicit dereference 13.14(11.1/1)
subtypes of the profile of a callable entity 13.14(14/3)
type caused by a range 13.14(12)
type caused by an expression 13.14(10/5)
type caused by the freezing of a subtype 13.14(15/5)
freezing points
entity 13.14(2)
Friday
in Ada.Calendar.Formatting 9.6.1(17/2)
From_Big_Integer
in Ada.Numerics.Big_Numbers.Big_Integers A.5.6(12/5), A.5.6(13/5)
From_Big_Real
in Ada.Numerics.Big_Numbers.Big_Reals A.5.7(13/5), A.5.7(14/5)
From_Quotient_String
in Ada.Numerics.Big_Numbers.Big_Reals A.5.7(19/5)
From_String
in Ada.Numerics.Big_Numbers.Big_Integers A.5.6(15/5)
in Ada.Numerics.Big_Numbers.Big_Reals A.5.7(16/5)
From_Universal_Image
in Ada.Numerics.Big_Numbers.Big_Integers A.5.6(16/5)
in Ada.Numerics.Big_Numbers.Big_Reals A.5.7(17/5), A.5.7(18/5)
FS
in Ada.Characters.Latin_1 A.3.3(6)
full access
object C.6(8.2/5)
type C.6(8.2/5)
full conformance
for discrete_subtype_definitions 6.3.1(24)
for expressions 6.3.1(19)
for known_discriminant_parts 6.3.1(23)
for profiles 6.3.1(18/3)
required 3.10.1(4/3), 6.3(4/5), 6.7(2.1/3), 6.8(4/3), 7.3(9/5), 8.3(12.3/5), 8.5.4(5/3), 9.5.2(14), 9.5.2(16), 9.5.2(17), 10.1.3(11/5), 10.1.3(12)
full constant declaration 3.3.1(6/5)
corresponding to a formal object of mode in 12.4(10/2)
full declaration 7.4(2/3)
full name
of a file A.16(47/2)
full stop 2.1(15/5)
full type 1.3.1(32/5), 3.2.1(8/2)
full type definition 3.2.1(8/2)
full view 1.3.1(33/5)
of a type 3.2.1(8/2)
Full_Access_Only aspect C.6(6.5/5)
Full_Name
in Ada.Directories A.16(15/5), A.16(39/2)
Full_Stop
in Ada.Characters.Latin_1 A.3.3(8)
full_type_declaration 3.2.1(3/3)
used 3.2.1(2), P.1
function 1.3.2(1/5), 6(1)
expression 6.8(3.1/5)
property 7.3.4(2/5)
stable property 7.3.4(7/5)
with a controlling access result 3.9.2(2/3)
with a controlling result 3.9.2(2/3)
function call
master of 3.10.2(10.1/3)
function instance 12.3(13)
function_call 6.4(3)

used 4.1(2/5), P.1
function_specification 6.1(4.2/2)
used 6.1(4/2), 6.8(2/4), P.1

G

gaps 13.3(52.d/2)
general access type 3.10(7/1), 3.10(8)
general_access_modifier 3.10(4)
used 3.10(3), P.1
generalized iterator 5.5.2(3/3)
generalized_indexing 4.1.6(10/3)
used 4.1(2/5), P.1
generalized_reference 4.1.5(4/3)
used 4.1(2/5), P.1
Generate_Deadlines pragma D.2.6(9.4/5), L(13.2/5)
generation
of an interrupt C.3(2)
Generator
in Ada.Numerics.Discrete_Random A.5.2(19)
in Ada.Numerics.Float_Random A.5.2(7)
generic actual 12.3(7/5)
generic actual parameter 12.3(7/5)
generic actual subtype 12.5(4)
generic actual type 12.5(4)
generic body 12.2(1)
generic contract issue 10.2.1(10/2), 12.3(11.y)
[*partial*] 3.2.4(29/3), 3.4(5.1/3), 3.7(10/3), 3.7.1(7/3), 3.9.1(3/2), 3.9.3(7/5), 3.9.4(17/2), 3.10.2(28.1/3), 3.10.2(32/5), 4.1.6(9/5), 4.2.1(16/5), 4.5.2(9.8/5), 4.6(24.24/5), 4.8(5.6/3), 4.9(37/2), 6.1.1(17.2/4), 6.3.1(17.a), 6.4.1(5.4/4), 6.4.1(6.3/4), 6.5.1(6/5), 7.3(8), 8.3(26/2), 8.3.1(7/2), 8.5.1(5.1/5), 8.5.4(4.3/2), 9.1(9.9/2), 9.4(11.13/2), 9.4(11.8/2), 9.5(17/3), 9.5(71/5), 9.5.2(13.4/2), 10.2.1(11.8/5), 10.2.1(11/5), 10.2.1(17/5), 12.4(8.3/5), 12.6(8.3/2), 13.1.1(18.7/5), 13.11.2(3.1/3), 13.11.4(23/3), 13.13.2(49/4), B.3.3(10/3), C.3.1(7/3), D.2.1(1.12/5), E.2.3(15.1/5), H.4.1(4/5), J.15.7(7/3), J.15.14(12/5)
generic contract model 12.3(1.a/5)
generic contract/private type contract analogy 7.3(19.a)
generic formal 12.1(9)
generic formal object 12.4(1)
generic formal package 12.7(1)
generic formal subprogram 12.6(1)
generic formal subtype 12.5(5)
generic formal type 12.5(5)
generic function 12.1(8/2)
generic instance 1.3.3(12/5)
generic package 12.1(8/2)
generic procedure 12.1(8/2)
generic subprogram 12.1(8/2)
generic unit 1.3.3(13/5), 12(1)
See also dispatching operation 3.9(1)
generic_actual_part 12.3(3)
used 12.3(2/3), 12.7(3/2), P.1
Generic_Array_Sort
child of Ada.Containers A.18.26(3/5)
generic_association 12.3(4)
used 12.3(3), 12.7(3.1/2), P.1
Generic_Bounded_Length
in Ada.Strings.Bounded A.4.4(4)
Generic_Complex_Arrays
child of Ada.Numerics G.3.2(2/5)
Generic_Complex_Elementary_Functions
child of Ada.Numerics G.1.2(2/5)
Generic_Complex_Types
child of Ada.Numerics G.1.1(2/5)
Generic_Constrained_Array_Sort
child of Ada.Containers A.18.26(7/5)
generic_declaration 12.1(2)
used 3.1(3/3), 10.1.1(5), P.1
Generic_Dispatching_Constructor
child of Ada.Tags 3.9(18.2/5)
Generic_Elementary_Functions
child of Ada.Numerics A.5.1(3/5)
generic_formal_parameter_declaration 12.1(6)
used 12.1(5), P.1
generic_formal_part 12.1(5)
used 12.1(3/3), 12.1(4), P.1
generic_instantiation 12.3(2/3)
used 3.1(3/3), 10.1.1(5), P.1
Generic_Keys
in Ada.Containers.Hashed_Sets A.18.8(50/5)
in Ada.Containers.Ordered_Sets A.18.9(62/5)
generic_package_declaration 12.1(4)
used 12.1(2), P.1
Generic_Real_Arrays
child of Ada.Numerics G.3.1(2/5)
generic_renaming_declaration 8.5.5(2/3)
used 8.5(2), 10.1.1(6), P.1
Generic_Sort
child of Ada.Containers A.18.26(9.2/5)
Generic_Sorting
in Ada.Containers.Doubly_Linked_-Lists A.18.3(47/5)
in Ada.Containers.Vectors A.18.2(75/5)
generic_subprogram_declaration 12.1(3/3)
used 12.1(2), P.1
Get
in Ada.Strings.Text_Buffers.Unbounded A.4.12(20/5)
in Ada.Text_IO A.10.1(41/5), A.10.1(47/5), A.10.1(54/5), A.10.1(55/5), A.10.1(59/5), A.10.1(60/5), A.10.1(65/5), A.10.1(67/5), A.10.1(70/5), A.10.1(72/5), A.10.1(75/5), A.10.1(77/5), A.10.1(81/5), A.10.1(83/5)
in Ada.Text_IO.Complex_IO G.1.3(6), G.1.3(8/5)
Get_CPU
in Ada.Interrupts C.3.2(10.1/3)
in System.Multiprocessors.Dispatching_Domains D.16.1(13/3)
Get_CPU_Set
in System.Multiprocessors.Dispatching_Domains D.16.1(9.3/4)
Get_Deadline
in Ada.Dispatching.EDF D.2.6(9/5)
Get_Dispatching_Domain
in System.Multiprocessors.Dispatching_Domains D.16.1(10/3)
Get_First_CPU
in System.Multiprocessors.Dispatching_Domains D.16.1(8/3)
Get_Immediate
in Ada.Text_IO A.10.1(44/5), A.10.1(45/5)
Get_Last_CPU
in System.Multiprocessors.Dispatching_Domains D.16.1(9/4)
Get_Last_Release_Time
in Ada.Dispatching.EDF D.2.6(9/5)
Get_Line
in Ada.Text_IO A.10.1(49.1/5), A.10.1(49/5)
in Ada.Text_IO.Bounded_IO A.10.11(8/2), A.10.11(9/2), A.10.11(10/2), A.10.11(11/2)
in Ada.Text_IO.Unbounded_IO A.10.12(8/2), A.10.12(9/2), A.10.12(10/2), A.10.12(11/2)
Get_Next_Entry
in Ada.Directories A.16(35/2)
Get_Priority
in Ada.Dynamic_Priorities D.5.1(5)
Get_Relative_Deadline
in Ada.Dispatching.EDF D.2.6(9/5)
Get_UTF_8
in Ada.Strings.Text_Buffers.Unbounded A.4.12(23/5)
Global aspect 6.1.2(11/5), 6.1.2(16/5)
global to 8.1(15)
global variable set 6.1.2(24/5)
Global'Class aspect 6.1.2(15/5)
global_aspect_definition 6.1.2(2/5)
used 13.1.1(4/5), P.1
global_aspect_element 6.1.2(3/5)
used 6.1.2(2/5), P.1
global_designator 6.1.2(7/5)
used 6.1.2(2/5), P.1
global_mode 6.1.2(4/5)
used 6.1.2(2/5), 6.1.2(3/5), P.1
global_name 6.1.2(8/5)
used 6.1.2(6/5), 6.1.2(7/5), P.1
global_set 6.1.2(6/5)

used 6.1.2(3/5), P.1
glyphs 2.1(15.a/5)
goto_statement 5.8(2)
used 5.1(4/2), P.1
govern a variant 3.8.1(20)
govern a variant_part 3.8.1(20)
grammar
ambiguous 1.1.4(14.a)
complete listing P.1(1/5)
cross reference P.2(1/3)
notation 1.1.4(3)
resolution of ambiguity 1.1.4(14.a), 8.6(3)
under Syntax heading 1.1.2(25)
graphic character
a category of Character A.3.2(23)
graphic symbols 2.1(15.a/5)
graphic_character 2.1(14/3)
used 2.5(2), 2.6(3), P.1
Graphic_Set
in Ada.Strings.Maps.Constants A.4.6(4)
Grave
in Ada.Characters.Latin_1 A.3.3(13)
greater than operator 4.5(1), 4.5.2(1)
greater than or equal operator 4.5(1), 4.5.2(1)
greater-than sign 2.1(15/5)
Greater_Than_Sign
in Ada.Characters.Latin_1 A.3.3(10)
Greatest_Common_Divisor
in Ada.Numerics.Big_Numbers.Big_Integers A.5.6(19/5)
Group_Budget
in Ada.Execution_Time.Group_Budgets D.14.2(4/3)
Group_Budget_Error
in Ada.Execution_Time.Group_Budgets D.14.2(11/2)
Group_Budget_Handler
in Ada.Execution_Time.Group_Budgets D.14.2(5/5)
Group_Budgets
child of Ada.Execution_Time D.14.2(3/5)
GS
in Ada.Characters.Latin_1 A.3.3(6)
guard 9.7.1(3)
used 9.7.1(2), P.1

H

handle
an exception 11(1/3)
an exception occurrence 11(1.a)
an exception occurrence 11.4(1), 11.4(7)
subpool 13.11.4(18/3)
handle an exception 1.3.5(4/5)
handled_sequence_of_statements 11.2(2)
used 5.6(2), 6.3(2/3), 6.5(2.2/3), 7.2(2/3), 9.1(6/3), 9.5.2(3), 9.5.2(5/5), P.1
handler 11.2(5.a)
execution timer D.14.1(13/2)
group budget D.14.2(14/2)
interrupt C.3(2)
termination C.7.3(8/3)
timing event D.15(10/2)
Handling
child of Ada.Characters A.3.2(2/5)
child of Ada.Wide_Characters A.3.5(3/5)
child of Ada.Wide_Wide_Characters A.3.6(1/3)
Has_Element
in Ada.Containers.Doubly_Linked_Lists A.18.3(9.1/5), A.18.3(9.2/5), A.18.3(51.6/5), A.18.3(69.4/5)
in Ada.Containers.Hashed_Maps A.18.5(6.1/5), A.18.5(6.2/5), A.18.5(37.7/5)
in Ada.Containers.Hashed_Sets A.18.8(6.1/5), A.18.8(6.2/5), A.18.8(59.6/5)
in Ada.Containers.Multiway_Trees A.18.10(12.1/5), A.18.10(12/5), A.18.10(70.6/5), A.18.10(92.1/5)
in Ada.Containers.Ordered_Maps A.18.6(7.1/5), A.18.6(7.2/5), A.18.6(51.8/5)
in Ada.Containers.Ordered_Sets A.18.9(7.1/5), A.18.9(7.2/5), A.18.9(74.6/5)
in Ada.Containers.Vectors A.18.2(11.1/5), A.18.2(11.2/5), A.18.2(79.6/5), A.18.2(97.4/5)
Has_Same_Storage attribute 13.3(73.2/4)
Hash
child of Ada.Strings A.4.9(2/5)
child of Ada.Strings.Bounded A.4.9(7/5)
child of Ada.Strings.Unbounded A.4.9(10/5)
Hash_Case_Insensitive
child of Ada.Strings A.4.9(11.2/5)
child of Ada.Strings.Bounded A.4.9(11.7/5)
child of Ada.Strings.Fixed A.4.9(11.5/3)
child of Ada.Strings.Unbounded A.4.9(11.10/5)
Hash_Type
in Ada.Containers A.18.1(4/2)
Hashed_Maps
child of Ada.Containers A.18.5(2/5)
Hashed_Sets
child of Ada.Containers A.18.8(2/5)
Head
in Ada.Strings.Bounded A.4.4(70), A.4.4(71)
in Ada.Strings.Fixed A.4.3(35), A.4.3(36)
in Ada.Strings.Unbounded A.4.5(65), A.4.5(66)
head (of a queue) D.2.1(5/2)
heap management
user-defined 13.11(1)
See also allocator 4.8(1)
Heart of Darkness 3.10.2(3.b/3)
held priority D.11(4/2)
heterogeneous input-output A.12.1(1)
hexadecimal
literal 2.4.2(1)
hexadecimal digit
a category of Character A.3.2(30)
hexadecimal literal 2.4.2(1)
Hexadecimal_Digit_Set
in Ada.Strings.Maps.Constants A.4.6(4)
hidden from all visibility 8.3(5), 8.3(14)
by lack of a with_clause 8.3(20/2)
for a declaration completed by a subsequent declaration 8.3(19)
for overridden declaration 8.3(15)
within the declaration itself 8.3(16)
hidden from direct visibility 8.3(5), 8.3(21)
by an inner homograph 8.3(22)
where hidden from all visibility 8.3(23)
hiding 8.3(5)
Hierarchical_File_Names
child of Ada.Directories A.16.1(3/5)
child of Ada.Wide_Directories A.16.2(2/5)
child of Ada.Wide_Wide_Directories A.16.2(3/5)
High_Order_First 13.5.3(2)
in Interfaces.COBOL B.4(25)
in System 13.7(15/2)
highest precedence operator 4.5.6(1)
highest_precedence_operator 4.5(7)
Hold
in Ada.Asynchronous_Task_Control D.11(3/5)
Holder
in Ada.Containers.Indefinite_Holders A.18.18(6/5)
homograph 8.3(8)
Hour
in Ada.Calendar.Formatting 9.6.1(24/2)
Hour_Number *subtype of* Natural
in Ada.Calendar.Formatting 9.6.1(20/2)
HT
in Ada.Characters.Latin_1 A.3.3(5)
HTJ
in Ada.Characters.Latin_1 A.3.3(17)
HTS
in Ada.Characters.Latin_1 A.3.3(17)
Hyphen
in Ada.Characters.Latin_1 A.3.3(8)
hyphen-minus 2.1(15/5)

I

i
  in Ada.Numerics.Generic_Complex_-Types G.1.1(5)
  in Interfaces.Fortran B.5(10)
identifier 2.3(2/2)
  used 2.8(2), 2.8(3/3), 2.8(21), 2.8(23), 3.1(4), 3.8(3/5), 4.1(3), 4.1.3(3), 4.1.4(3/2), 4.5.10(4/5), 5.5(2), 5.6(2), 6.1(5), 7.1(3/3), 7.2(2/3), 9.1(4), 9.1(6/3), 9.4(4), 9.4(7/3), 9.5.2(3), 9.5.2(5/5), 9.10.1(3/5), 11.4.2(6.1/3), 11.4.2(6/2), 11.5(4.1/2), 11.5(4/2), 13.1.1(3/3), 13.1.1(4/5), 13.12(4/2), 13.12(11/3), D.2.2(2), D.2.2(2.2/2), D.3(3), D.3(4), D.3(4.a), D.4(3), D.4(4/5), D.4.1(3/5), H.6(3/2), J.10(3/2), J.15.5(2/3), J.15.5(3/3), J.15.5(4/3), L(1.1/5), L(2.3/2), L(2.4/3), L(6.2/5), L(8.1/3), L(13.1/3), L(14.1/3), L(20), L(21), L(23), L(25.1/2), L(27.2/2), L(27.3/3), L(29), L(36), L(37), L(37.3/2), M.2(98), P.1
identifier specific to a pragma 2.8(10/3)
identifier_extend 2.3(3.1/3)
  used 2.3(2/2), P.1
identifier_start 2.3(3/2)
  used 2.3(2/2), P.1
Identity
  in Ada.Strings.Maps A.4.2(22)
  in Ada.Strings.Wide_Maps A.4.7(22)
  in Ada.Strings.Wide_Wide_Maps A.4.8(22/2)
Identity attribute 11.4.1(9), C.7.1(12)
idle task D.11(4/2)
IEEE floating point arithmetic B.2(10.a), G.2.2(11.a/5)
IEEE_Float_32 B.2(10.a)
IEEE_Float_64 B.2(10.a)
if expression 4.5.7(1/3)
if_expression 4.5.7(3/3)
  used 4.5.7(2/3), P.1
if_statement 5.3(2)
  used 5.1(5/5), P.1
illegal
  construct 1.1.2(27)
  partition 1.1.2(29)
Im
  in Ada.Numerics.Generic_Complex_-Arrays G.3.2(7/2), G.3.2(27/2)
  in Ada.Numerics.Generic_Complex_-Types G.1.1(6)
image
  of a value 4.10(1/5)
  in Ada.Calendar.Formatting 9.6.1(35/2), 9.6.1(37/2)
  in Ada.Numerics.Discrete_Random A.5.2(26)
  in Ada.Numerics.Float_Random A.5.2(14)
  in Ada.Task_Identification C.7.1(3/5)
  in Ada.Text_IO.Editing F.3.3(13)
Image attribute 4.10(34/5), 4.10(40/5)
Imaginary
  in Ada.Numerics.Generic_Complex_-Types G.1.1(4/5)
Imaginary *subtype of* Imaginary
  in Interfaces.Fortran B.5(10)
immediate scope
  of (a view of) an entity 8.2(11)
  of a declaration 8.2(2)
  of a pragma 8.2(12/4)
Immediate_Reclamation restriction H.4(10)
immediately enclosing 8.1(13)
immediately visible 8.3(4), 8.3(21)
immediately within 8.1(13)
immutably limited 7.5(8.1/3)
implementation 1.1.3(1.a)
implementation advice 1.1.2(37)
  summary of advice M.3(1/5)
implementation defined 1.1.3(18)
  summary of characteristics M.2(1/5)
implementation permissions 1.1.2(36)
implementation requirements 1.1.2(33)
implementation-dependent
  See unspecified 1.1.3(18)
implemented
  by a protected entry 9.4(11.1/3)
  by a protected subprogram 9.4(11.1/3)
  by a task entry 9.1(9.2/3)
implicit conversion
  legality 8.6(27.1/4)
implicit declaration 3.1(5)
implicit initial values
  for a subtype 3.3.1(10)
implicit subtype conversion 4.6(59), 4.6(60)
  Access attribute 3.10.2(30/5)
  access discriminant 3.7(27/2)
  array bounds 4.6(38)
  array index 4.1.1(7)
  assignment to view conversion 4.6(55/5)
  assignment_statement 5.2(11/5)
  bounds of a decimal fixed point type 3.5.9(16)
  bounds of a fixed point type 3.5.9(14)
  bounds of a range 3.5(9), 3.6(18)
  choices of aggregate 4.3.3(22)
  component defaults 3.3.1(13/5)
  default value of a scalar 3.3.1(11.1/5)
  delay expression 9.6(20)
  derived type discriminants 3.4(21)
  discriminant values 3.7.1(12/5)
  entry index 9.5.2(24/5)
  expressions in aggregate 4.3.1(19/5)
  expressions of aggregate 4.3.3(23)
  function return 6.5(5.12/5), 6.5(6/2)
  generic formal object of mode in 12.4(11)
  inherited enumeration literal 3.4(29)
  initialization expression 3.3.1(17/5)
  initialization expression of allocator 4.8(7/2)
  Interrupt_Priority aspect D.1(17/4), D.3(6.1/3)
  named number value 3.3.2(6)
  operand of concatenation 4.5.3(9)
  parameter passing 6.4.1(10), 6.4.1(11), 6.4.1(17)
  Priority aspect D.1(17/4), D.3(6.1/3)
  qualified_expression 4.7(4/5)
  reading a view conversion 4.6(56/5)
  result of inherited function 3.4(27/2)
implicit_dereference 4.1(6)
  used 4.1(4), P.1
Implicit_Dereference aspect 4.1.5(2/3)
implicitly composed
  aspect 13.1.1(18.3/5)
Import aspect B.1(1/3)
Import pragma J.15.5(2/3), L(14.1/3)
imported entity B.1(23/3)
in (membership test) 4.5(1), 4.5.2(2/3)
In_Range
  in Ada.Numerics.Big_Numbers.Big_Integers A.5.6(10/5)
  in Ada.Numerics.Big_Numbers.Big_Reals A.5.7(12/5)
inaccessible partition E.1(7)
inactive
  a task state 9(10/5)
Include
  in Ada.Containers.Hashed_Maps A.18.5(22/5)
  in Ada.Containers.Hashed_Sets A.18.8(21/5)
  in Ada.Containers.Ordered_Maps A.18.6(21/5)
  in Ada.Containers.Ordered_Sets A.18.9(20/5)
included
  one execution by another 11.4(2.a)
  one range in another 3.5(4)
incompatibilities with Ada 2012 5.5.2(17.b/4)

incompatibilities with Ada 2005 1.1.2(39.cc/3), 2.3(8.d/3), 2.9(3.e/3), 3.4(38.m/3), 3.5.1(16.c/3), 3.7(37.l/3), 3.8.1(29.e/3), 3.9(33.j/3), 3.9.3(17.j/3), 3.10.2(41.p/3), 4.3.1(31.e/3), 4.3.2(13.e/3), 4.5.2(39.k/3), 4.8(20.o/3), 4.9.1(10.d/3), 6.3.1(26.h/3), 6.4.1(19.e/3), 6.5(28.q/3), 8.5.1(10.g/3), 8.5.1(10.i/5), 8.6(34.v/3), 9.4(35.g/3), 10.1.2(32.i/3), 10.2.1(28.k/3), 11.4.2(28.b/3), 12.5.1(28.i/3), 12.5.4(13.e/3), 12.6(24.d/3), 12.7(25.e/3), 13.1(29.p/5), 13.3(86.n/3), 13.13.2(60.u/3), A.3.2(60.d/3), A.4.3(109.d/3), A.4.4(106.i/3), A.4.5(88.e/3), A.16(131.e/3), A.17(33.c/3), A.18.1(8.d/3), A.18.2(264.c/3), A.18.3(164.c/3), A.18.5(62.d/3), A.18.6(95.e/3), A.18.8(88.d/3), A.18.9(116.d/3), B.1(51.d/3), B.3(84.i/3), B.3.3(32.b/3), C.3.1(25.e/3), C.3.2(28.b/3), C.6(26.e/3), C.7.1(21.d/3), D.2.1(17.b/3), D.7(22.g/3), D.14(29.c/3), E.2.2(20.j/5), E.2.3(20.e/3), J.15.7(8.a/3)
incompatibilities with Ada 2012 1.1.2(39.pp/5), 2.3(8.f/5), 2.9(3.g/5), 3.3(26.o/5), 3.5.9(28.e/4), 3.9.3(17.o/5), 3.10.2(41.hh/4), 4.1.3(19.m/5), 4.1.5(16.b/4), 4.1.6(23.b/5), 4.1.6(23.c/5), 4.3(6.l/5), 4.4(15.g/4), 4.5.2(39.q/5), 4.9(44.z/5), 5.5.1(22.c/4), 6.1.1(44.e/5), 6.3.1(26.l/5), 6.4.1(19.l/5), 6.5.1(11.c/5), 8.5.4(21.h/5), 9.5(72.e/5), 9.5.4(20.e/4), 10.2.1(28.t/5), 12.5.1(28.o/4), 12.6(24.g/5), 13.1(29.u/4), 13.1.1(39.c/5), 13.1.1(39.d/4), 13.13.2(60.cc/5), A(4.k/5), A.3.2(60.h/5), A.3.5(64.b/5), A.8.1(17.c/5), A.8.4(20.b/5), A.10.1(86.f/5), A.12.1(36.h/5), A.18.2(264.n/5), A.18.3(164.n/5), A.18.5(62.g/5), A.18.6(95.i/5), A.18.8(88.h/5), A.18.9(116.i/5), A.18.10(233.e/5), A.18.18(74.d/5), A.18.19(17.c/5), A.18.20(20.c/5), A.18.21(22.c/5), A.18.22(19.c/5), A.18.23(21.b/5), A.18.24(18.b/5), A.18.25(20.c/5), B.1(51.f/5), B.3(84.j/5), D.13(13.c/4), D.16.1(34.d/5), E.2.1(11.f/4), E.2.3(20.i/4), E.2.3(20.j/5), H.4(28.l/5)
incompatibilities with Ada 83 1.1.2(39.e), 2.8(19.c), 2.9(3.a), 3.2.2(15.a), 3.2.3(8.a), 3.4(38.b), 3.5(63.a/1), 3.5.2(11.b), 3.6.3(8.b), 4.2(14.a), 4.3.3(49.a.1/1), 4.5.5(35.a.1/2), 4.6(71.a), 4.8(20.a/1), 4.9(44.m), 5.4(18.a.1/1), 6.5(28.a/2), 7.1(17.a/5), 8.6(34.a), 12.3(29.b), 12.5.1(28.a), 12.5.3(16.a), 12.5.4(13.a), 13.1(29.a), 13.14(20.a), A.5.3(72.d), A.5.4(4.a), A.8.1(17.a), A.10.1(86.a), C.6(26.b), J.15.1(6.a/3)
incompatibilities with Ada 95 1.1.2(39.p/2), 2.9(3.c/2), 3.7.1(15.c/2), 3.9(33.d/3), 3.9.2(24.b/2), 3.10(26.e/2), 3.10.1(23.h/2), 3.10.2(41.b/2), 4.3(6.e/2), 4.3.2(13.b/2), 4.5.5(35.d/2), 4.6(71.j/2), 4.8(20.g/2), 5.2(29.d/2), 6.5(28.g/2), 8.3(29.s/2), 8.5.1(10.b/2), 8.6(34.n/2), 9.7.2(7.b/3), 10.1.2(32.f/2), 10.2.1(28.e/2), 11.4.1(20.e/3), 13.5.1(32.d/2), 13.5.2(5.c/2), 13.11(43.d/2), 13.11.3(9.b/3), 13.14(20.p/2), 13.14(20.t/3), A.4.1(6.a/3), A.4.3(109.a/3), A.4.4(106.f/3), A.4.5(88.b/2), A.4.7(49.a/2), A.5.2(62.a/2), A.8.1(17.b/2), A.8.4(20.a/2), A.10.1(86.c/2), A.10.7(26.a/3), A.12.1(36.b/2), B.3(84.a/3), C.3.1(25.a/2), D.7(22.a/2), D.8(51.a/3), E.2.2(20.b/3), E.2.3(20.b/3), E.5(30.a/2)
incompatiblities with Ada 2012 7.3.2(41.h/5), D.2.6(34.e/5)
incomplete type 1.3.1(35/5), 3.2(4.1/2), 3.10.1(2.1/4)
incomplete view 1.3.1(37/5), 3.10.1(2.1/4)
  tagged 3.10.1(2.1/4)
incomplete_type_declaration 3.10.1(2/2)
  used 3.2.1(2), P.1
inconsistencies with Ada 2005 1.1.2(39.z/3), 2.1(19.g/3), 3.5(63.l/3), 3.9(33.i/3), 4.3.2(13.d/3), 4.3.3(49.i/3), 4.5.2(39.i/3), 4.6(71.u/3), 6.4.1(19.d/3), 6.5(28.n/3), 7.6.1(24.cc/3), 8.5.4(21.f.1/3), 9.6.1(91.c/3), 13.3(86.l/3), A.10.5(52.a/3), A.10.8(26.a/3), A.16(131.d/3), A.18.3(164.b/3), D.14.2(38.b/3)
inconsistencies with Ada 2012 1.1.2(39.mm/5), 3.2.4(52.c/5), 3.3.1(33.n/5), 3.10.2(41.aa/5), 4.3.3(49.k/4), 4.4(15.f/4), 4.5.2(39.p/4), 4.7(10.b/5), 6.1.1(44.b/4), 6.4.1(19.k/5), 6.5(28.x/5), 7.3.2(41.b/4), 8.6(34.y/4), 9.5(72.d/5), 9.5.4(20.c/4), 9.6.1(91.e/5), 12.5(16.j/5), 13.11.2(19.e/4), A.18.2(264.l/5), A.18.3(164.m/5), A.18.4(84.k/5), A.18.7(105.k/5), A.18.10(233.d/5), A.18.11(13.b/4), A.18.12(12.b/4), A.18.13(13.b/4), A.18.14(13.b/4), A.18.15(5.e/4), A.18.16(5.e/4), A.18.17(12.b/4), A.18.18(74.c/4), A.18.19(17.b/5), A.18.20(20.b/5), A.18.21(22.b/5), A.18.22(19.b/5), A.18.25(20.b/5), A.19(12.c/4), C.3.1(25.f/5), C.6(26.j/5), D.2.1(17.d/5), D.16.1(34.c/4)
inconsistencies with Ada 83 1.1.2(39.b), 3.4(38.a), 3.5.2(11.a), 3.5.7(22.a), 3.5.9(28.a), 3.6.3(8.a), 3.7.1(15.a), 4.5.3(14.a), 4.5.6(13.a.1/1), 9.6(40.a), 11.1(8.a), 12.3(29.a), 13.3(58.a.1/2), A.6(1.a), G.2.1(16.c), G.2.3(27.b)
inconsistencies with Ada 95 1.1.2(39.m/2), 3.3.1(33.f/2), 3.5.2(11.h/2), 3.6.3(8.g/2), 3.9(33.b/2), 3.10(26.c/2), 4.8(20.f/2), 4.9(44.s/2), 6.5(28.f.1/3), 7.6.1(24.v.1/3), 9.6(40.e/2), 11.4.1(20.b/2), 13.13.2(60.g/2), A.4.4(106.e/2), A.12.1(36.a/3), B.3.1(60.a/2)
Increase_Indent
  in Ada.Strings.Text_Buffers A.4.12(15/5)
Increment
  in Interfaces.C.Pointers B.3.2(11/3)
indefinite subtype 3.3(23/5), 3.7(26)
Indefinite_Doubly_Linked_Lists
  child of Ada.Containers A.18.12(2/3)
Indefinite_Hashed_Maps
  child of Ada.Containers A.18.13(2/3)
Indefinite_Hashed_Sets
  child of Ada.Containers A.18.15(2/3)
Indefinite_Holders
  child of Ada.Containers A.18.18(5/5)
Indefinite_Multiway_Trees
  child of Ada.Containers A.18.17(2/3)
Indefinite_Ordered_Maps
  child of Ada.Containers A.18.14(2/3)
Indefinite_Ordered_Sets
  child of Ada.Containers A.18.16(2/3)
Indefinite_Vectors
  child of Ada.Containers A.18.11(2/3)
Independent aspect C.6(6.3/3)
Independent pragma J.15.8(4/3), L(14.2/3)
independent subprogram 11.6(6/5)
Independent_Components aspect C.6(6.10/3)

Independent_Components pragma J.15.8(7/3), L(14.3/3)
independently addressable 9.10(1/5)
  specified C.6(8.1/4)
index
  of an array 3.6(9.a)
  of an element of an open direct file A.8(3)
  in Ada.Direct_IO A.8.4(15)
  in Ada.Streams.Stream_IO A.12.1(23)
  in Ada.Strings.Bounded A.4.4(43.1/2), A.4.4(43.2/2), A.4.4(44), A.4.4(45), A.4.4(45.1/2), A.4.4(46)
  in Ada.Strings.Fixed A.4.3(8.1/2), A.4.3(8.2/2), A.4.3(9), A.4.3(10), A.4.3(10.1/2), A.4.3(11)
  in Ada.Strings.Unbounded A.4.5(38.1/2), A.4.5(38.2/2), A.4.5(39), A.4.5(40), A.4.5(40.1/2), A.4.5(41)
Index attribute 6.1.1(30.2/5)
index parameter
  of an array aggregate 4.3.3(6.1/5)
index range 3.6(13)
index subtype 3.6(9)
index type 3.6(9)
  container aggregate 4.3.5(25/5)
  indexed aggregate 4.3.5(25/5)
Index_Check 11.5(14)
  [*partial*] 4.1.1(7), 4.1.2(7), 4.3.3(20.4/5), 4.3.3(29/3), 4.3.3(30), 4.3.4(20/5), 4.5.3(8), 4.6(51/5), 4.7(4/5), 4.8(10/2)
index_constraint 3.6.1(2)
  used 3.2.2(7), P.1
Index_Error
  in Ada.Strings A.4.1(5)
Index_Non_Blank
  in Ada.Strings.Bounded A.4.4(46.1/2), A.4.4(47)
  in Ada.Strings.Fixed A.4.3(11.1/2), A.4.3(12)
  in Ada.Strings.Unbounded A.4.5(41.1/2), A.4.5(42)
index_subtype_definition 3.6(4)
  used 3.6(3), P.1
indexable container object 4.1.6(5/3)
indexable container type 1.3.1(39/5), 4.1.6(5/3)
indexed aggregate 4.3.5(25/5)
indexed_component 4.1.1(2)
  used 4.1(2/5), P.1
indexing
  constant 4.1.6(12/3)
  variable 4.1.6(16/3)
individual membership test 4.5.2(26.1/3)
indivisible C.6(10/4)
inferable discriminants B.3.3(20/2)
Information
  child of Ada.Directories A.16(124/2)
  child of Ada.Wide_Directories A.16.2(2/5)
  child of Ada.Wide_Wide_Directories A.16.2(3/5)
information hiding
  See package 7(1)
  See private types and private extensions 7.3(1)
information systems C(1), F(1)
informative 1.1.2(18)
inherently mutable object 3.3(13/3)
inheritance
  See derived types and classes 3.4(1/2)
  See also tagged types and type extension 3.9(1)
inherited
  from an ancestor type 3.4.1(11)
inherited component 3.4(11), 3.4(12)
inherited discriminant 3.4(11)
inherited entry 3.4(12)
inherited protected subprogram 3.4(12)
inherited subprogram 3.4(17/2)
initial value
  reduction attribute 4.5.10(24/5)
Initial_Directory
  in Ada.Directories.Hierarchical_File_Names A.16.1(12/3)
initialization
  of a protected object 9.4(14)
  of a protected object C.3.1(10/3), C.3.1(11/3)
  of a task object 9.1(12/1), J.7.1(7)
  of an object 3.3.1(18/2)
initialization expression 3.3.1(1/3), 3.3.1(4)
Initialize 7.6(2)
  in Ada.Finalization 7.6(6/2), 7.6(8/2)
initialized allocator 4.8(4)
initialized by default 3.3.1(18/2)
Inline aspect 6.3.2(5.1/3)
Inline pragma J.15.1(2/3), L(15.1/3)
innermost dynamically enclosing 11.4(2)
input A.6(1/5)
Input aspect 13.13.2(38/5)
Input attribute 13.13.2(22), 13.13.2(32)
Input clause 13.3(7/2), 13.13.2(38/5)
Input'Class aspect 13.13.2(38/5)
input-output
  unspecified for access types A.7(6)
Insert
  in Ada.Containers.Doubly_Linked_Lists A.18.3(19/5), A.18.3(20/5), A.18.3(21/5)
  in Ada.Containers.Hashed_Maps A.18.5(19/5), A.18.5(20/5), A.18.5(21/5)
  in Ada.Containers.Hashed_Sets A.18.8(19/5), A.18.8(20/5)
  in Ada.Containers.Ordered_Maps A.18.6(18/5), A.18.6(19/5), A.18.6(20/5)
  in Ada.Containers.Ordered_Sets A.18.9(18/5), A.18.9(19/5)
  in Ada.Containers.Vectors A.18.2(39/5), A.18.2(40/5), A.18.2(41/5), A.18.2(42/5), A.18.2(43/5)
  in Ada.Strings.Bounded A.4.4(60), A.4.4(61)
  in Ada.Strings.Fixed A.4.3(25), A.4.3(26)
  in Ada.Strings.Unbounded A.4.5(55), A.4.5(56)
Insert_Child
  in Ada.Containers.Multiway_Trees A.18.10(48/5), A.18.10(49/5), A.18.10(50/5)
Insert_Space
  in Ada.Containers.Vectors A.18.2(48/5), A.18.2(49/5)
Insert_Vector
  in Ada.Containers.Vectors A.18.2(36/5), A.18.2(37/5), A.18.2(38/5)
inspectable object H.3.2(5/2)
inspection point H.3.2(5/2)
Inspection_Point pragma H.3.2(3), L(16)
instance
  of a generic function 12.3(13)
  of a generic package 12.3(13)
  of a generic procedure 12.3(13)
  of a generic subprogram 12.3(13)
  of a generic unit 12.3(1)
instructions for comment submission 0.2(57.22/5)
int
  in Interfaces.C B.3(7)
Integer 3.5.4(11), 3.5.4(21)
  in Standard A.1(12)
integer literal 2.4(1)
integer literals 3.5.4(14), 3.5.4(30)
integer type 1.3.1(40/5), 3.5.4(1)
Integer_Address
  in System.Storage_Elements 13.7.1(10/3)
Integer_Arithmetic
  child of System.Atomic_Operations C.6.4(3/5)
Integer_IO
  in Ada.Text_IO A.10.1(52)
Integer_Literal aspect 4.2.1(3/5)
Integer_N B.2(8)
Integer_Text_IO
  child of Ada A.10.8(21)
integer_type_definition 3.5.4(2)
  used 3.2.1(4/2), P.1
Integer_Wide_Text_IO
  child of Ada A.11(2/2)
Integer_Wide_Wide_Text_IO
  child of Ada A.11(3/2)
interaction
  between tasks 9(1/5)
interface 3.9.4(4/2)
  limited 3.9.4(5/2)
  nonlimited 3.9.4(5/2)
  protected 3.9.4(5/2)
  synchronized 3.9.4(5/2)
  task 3.9.4(5/2)
  type 3.9.4(4/2)
interface to assembly language C.1(4/3)
interface to C B.3(1/4)

interface to COBOL B.4(1/3)
interface to Fortran B.5(1/3)
interface to other languages B(1)
interface type 1.3.1(42/5)
Interface_Ancestor_Tags
in Ada.Tags 3.9(7.4/2)
interface_list 3.9.4(3/2)
used 3.4(2/2), 3.9.4(2/2), 7.3(3/3), 9.1(2/3), 9.1(3/5), 9.4(2/3), 9.4(3/3), 12.5.1(3/5), P.1
interface_type_definition 3.9.4(2/2)
used 3.2.1(4/2), 12.5.5(2/2), P.1
Interfaces B.2(3/5)
Interfaces.C B.3(4/5)
Interfaces.C.Pointers B.3.2(4/5)
Interfaces.C.Strings B.3.1(3/5)
Interfaces.COBOL B.4(7/5)
Interfaces.Fortran B.5(4/5)
Interfaces_Assertion_Check 11.5(23.7/5)
interfacing aspect B.1(0.1/3)
interfacing pragma J.15.5(1/3)
Convention J.15.5(1/3)
Export J.15.5(1/3)
Import J.15.5(1/3)
internal call 9.5(3/3)
internal code 13.4(7)
internal node
of a tree A.18.10(2/4)
internal requeue 9.5(7)
Internal_Tag
in Ada.Tags 3.9(7/2)
interpretation
of a complete context 8.6(10)
of a constituent of a complete context 8.6(15)
overload resolution 8.6(14)
interrupt C.3(2)
example using asynchronous_select 9.7.4(10), 9.7.4(12)
interrupt entry J.7.1(5)
interrupt handler C.3(2)
Interrupt_Clocks_Supported
in Ada.Execution_Time D.14(9.1/3)
Interrupt_Handler aspect C.3.1(6.2/3)
Interrupt_Handler pragma J.15.7(2/3), L(17.1/3)
Interrupt_Id
in Ada.Interrupts C.3.2(2/5)
Interrupt_Priority aspect D.1(6.3/3)
Interrupt_Priority pragma J.15.11(4/3), L(18.1/3)
Interrupt_Priority *subtype of* Any_Priority
in System 13.7(16)
Interrupts
child of Ada C.3.2(2/5)
child of Ada.Execution_Time D.14.3(3/5)
Intersection
in Ada.Containers.Hashed_Sets A.18.8(29/5), A.18.8(30/5)
in Ada.Containers.Ordered_Sets A.18.9(30/5), A.18.9(31/5)
intertask communication 9.5(1)
See also task 9(1/5)
Intrinsic calling convention 6.3.1(4)
invalid cursor
of a list container A.18.3(153/2)
of a map A.18.4(76/2)
of a set A.18.7(97/2)
of a tree A.18.10(222/3)
of a vector A.18.2(248/2)
invalid representation 13.9.1(9)
invariant 1.3.1(44/5), 7.3.2(2/3)
class-wide 7.3.2(3/5)
invariant check 7.3.2(9/4)
invariant expression 7.3.2(2/3)
Inverse
in Ada.Numerics.Generic_Complex_Arrays G.3.2(46/2)
in Ada.Numerics.Generic_Real_Arrays G.3.1(24/2)
Inverted_Exclamation
in Ada.Characters.Latin_1 A.3.3(21/3)
Inverted_Question
in Ada.Characters.Latin_1 A.3.3(22)
involve an inner product
complex G.3.2(56/2)
real G.3.1(34/2)
IO_Assertion_Check 11.5(23.8/5)
IO_Exceptions
child of Ada A.13(3/5)
IS1
in Ada.Characters.Latin_1 A.3.3(16)
IS2
in Ada.Characters.Latin_1 A.3.3(16)
IS3
in Ada.Characters.Latin_1 A.3.3(16)
IS4
in Ada.Characters.Latin_1 A.3.3(16)
Is_A_Group_Member
in Ada.Execution_Time.Group_Budgets D.14.2(8/2)
Is_Abstract
in Ada.Tags 3.9(7.5/3)
Is_Alphanumeric
in Ada.Characters.Handling A.3.2(4/5)
in Ada.Wide_Characters.Handling A.3.5(12/3)
Is_Ancestor_Of
in Ada.Containers.Multiway_Trees A.18.10(21.2/5)
Is_Attached
in Ada.Interrupts C.3.2(5)
Is_Basic
in Ada.Characters.Handling A.3.2(4/5)
in Ada.Wide_Characters.Handling A.3.5(8.1/5)
Is_Callable
in Ada.Task_Identification C.7.1(4/3)
Is_Character
in Ada.Characters.Conversions A.3.4(3/2)
Is_Control
in Ada.Characters.Handling A.3.2(4/5)
in Ada.Wide_Characters.Handling A.3.5(5/3)
Is_Current_Directory_Name
in Ada.Directories.Hierarchical_File_Names A.16.1(7/3)
Is_Decimal_Digit
in Ada.Characters.Handling A.3.2(4/5)
in Ada.Wide_Characters.Handling A.3.5(10/3)
Is_Descendant_At_Same_Level
in Ada.Tags 3.9(7.1/2)
Is_Digit
in Ada.Characters.Handling A.3.2(4/5)
in Ada.Wide_Characters.Handling A.3.5(9/3)
Is_Empty
in Ada.Containers.Doubly_Linked_Lists A.18.3(12/5)
in Ada.Containers.Hashed_Maps A.18.5(11/5)
in Ada.Containers.Hashed_Sets A.18.8(13/5)
in Ada.Containers.Indefinite_Holders A.18.18(10/5)
in Ada.Containers.Multiway_Trees A.18.10(16/5)
in Ada.Containers.Ordered_Maps A.18.6(10/5)
in Ada.Containers.Ordered_Sets A.18.9(12/5)
in Ada.Containers.Vectors A.18.2(23/5)
Is_Full_Name
in Ada.Directories.Hierarchical_File_Names A.16.1(8/3)
Is_Graphic
in Ada.Characters.Handling A.3.2(4/5)
in Ada.Wide_Characters.Handling A.3.5(19/3)
Is_Held
in Ada.Asynchronous_Task_Control D.11(3/5)
Is_Hexadecimal_Digit
in Ada.Characters.Handling A.3.2(4/5)
in Ada.Wide_Characters.Handling A.3.5(11/3)
Is_In
in Ada.Strings.Maps A.4.2(13)
in Ada.Strings.Wide_Maps A.4.7(13)
in Ada.Strings.Wide_Wide_Maps A.4.8(13/2)
Is_ISO_646
in Ada.Characters.Handling A.3.2(10)
Is_Leaf
in Ada.Containers.Multiway_Trees A.18.10(21.1/5), A.18.10(21/5)
Is_Letter
in Ada.Characters.Handling A.3.2(4/5)
in Ada.Wide_Characters.Handling A.3.5(6/3)
Is_Line_Terminator
in Ada.Characters.Handling A.3.2(4/5)
in Ada.Wide_Characters.Handling A.3.5(14/3)

Is_Lock_Free
  in System.Atomic_Operations.Exchange C.6.2(8/5)
  in System.Atomic_Operations.Integer_Arithmetic C.6.4(8/5)
  in System.Atomic_Operations.Modular_Arithmetic C.6.5(8/5)
  in System.Atomic_Operations.Test_And_Set C.6.3(7/5)
Is_Lower
  in Ada.Characters.Handling A.3.2(4/5)
  in Ada.Wide_Characters.Handling A.3.5(7/3)
Is_Mark
  in Ada.Characters.Handling A.3.2(4/5)
  in Ada.Wide_Characters.Handling A.3.5(15/3)
Is_Member
  in Ada.Execution_Time.Group_Budgets D.14.2(8/2)
Is_NFKC
  in Ada.Characters.Handling A.3.2(4/5)
  in Ada.Wide_Characters.Handling A.3.5(18.1/5)
Is_Nul_Terminated
  in Interfaces.C B.3(24), B.3(35), B.3(39.16/2), B.3(39.7/2)
Is_Open
  in Ada.Direct_IO A.8.4(10)
  in Ada.Sequential_IO A.8.1(10)
  in Ada.Streams.Stream_IO A.12.1(12)
  in Ada.Text_IO A.10.1(13)
Is_Other_Format
  in Ada.Characters.Handling A.3.2(4/5)
  in Ada.Wide_Characters.Handling A.3.5(16/3)
Is_Parent_Directory_Name
  in Ada.Directories.Hierarchical_File_Names A.16.1(6/3)
Is_Punctuation_Connector
  in Ada.Characters.Handling A.3.2(4/5)
  in Ada.Wide_Characters.Handling A.3.5(17/3)
Is_Relative_Name
  in Ada.Directories.Hierarchical_File_Names A.16.1(9/3)
Is_Reserved
  in Ada.Interrupts C.3.2(4)
Is_Root
  in Ada.Containers.Multiway_Trees A.18.10(20.1/5), A.18.10(20/5)
Is_Root_Directory_Name
  in Ada.Directories.Hierarchical_File_Names A.16.1(5/3)
Is_Round_Robin
  in Ada.Dispatching.Round_Robin D.2.5(4/5)
Is_Simple_Name
  in Ada.Directories.Hierarchical_File_Names A.16.1(4/3)
Is_Sorted
  in Ada.Containers.Doubly_Linked_Lists A.18.3(48/2)
  in Ada.Containers.Vectors A.18.2(76/2)
Is_Space
  in Ada.Characters.Handling A.3.2(4/5)
  in Ada.Wide_Characters.Handling A.3.5(18/3)
Is_Special
  in Ada.Characters.Handling A.3.2(4/5)
  in Ada.Wide_Characters.Handling A.3.5(13/3)
Is_Split
  in Ada.Iterator_Interfaces 5.5.1(4.3/5)
Is_String
  in Ada.Characters.Conversions A.3.4(3/2)
Is_Subset
  in Ada.Containers.Hashed_Sets A.18.8(39/2)
  in Ada.Containers.Ordered_Sets A.18.9(40/2)
  in Ada.Strings.Maps A.4.2(14)
  in Ada.Strings.Wide_Maps A.4.7(14)
  in Ada.Strings.Wide_Wide_Maps A.4.8(14/2)
Is_Terminated
  in Ada.Task_Identification C.7.1(4/3)
Is_Upper
  in Ada.Characters.Handling A.3.2(4/5)
  in Ada.Wide_Characters.Handling A.3.5(8/3)
Is_Valid
  in Ada.Numerics.Big_Numbers.Big_Integers A.5.6(4/5)
  in Ada.Numerics.Big_Numbers.Big_Reals A.5.7(4/5)
Is_Wide_Character
  in Ada.Characters.Conversions A.3.4(3/2)
Is_Wide_String
  in Ada.Characters.Conversions A.3.4(3/2)
ISO 1989:2002 1.2.1(4/5)
ISO 3166-1:2020 1.2(4.1/5)
ISO 639-3:2007 1.2(1.1/3)
ISO 8601-1:2019 1.2.1(6/5)
ISO/IEC 10646:2020 1.2(8/5), 3.5.2(2/5), 3.5.2(3/5), 3.5.2(4/5)
ISO/IEC 14882:2020 1.2.1(9/5)
ISO/IEC 1539-1:2018 1.2.1(3/5)
ISO/IEC 60559:2020 1.2.1(10/5), G.2.2(11.a/5)
ISO/IEC 6429:1992 1.2.1(5/5)
ISO/IEC 646:1991 1.2.1(2/5)
ISO/IEC 8859-1:1998 1.2.1(7/5)
ISO/IEC 9899:2018 1.2.1(8/5)
ISO_646 *subtype of* Character
  in Ada.Characters.Handling A.3.2(9)
ISO_646_Set
  in Ada.Strings.Maps.Constants A.4.6(4)
issue
  an entry call 9.5.3(8)
italics
  formal parameters of attribute functions 3.5(18.a)
  implementation-defined 1.1.3(5.c)
  nongraphic characters 3.5.2(2/5)
  pseudo-names of anonymous types 3.2.1(7/2), A.1(2)
  syntax rules 1.1.4(14)
  terms introduced or defined 1.3(1/5)
italics, like this 1(2.ll)
iterable container object 5.5.1(11/5)
iterable container object for a loop 5.5.2(12/5)
iterable container type 1.3.1(45/5), 5.5.1(11/5)
Iterate
  in Ada.Containers.Doubly_Linked_Lists A.18.3(45/5), A.18.3(46.1/5), A.18.3(46.2/5)
  in Ada.Containers.Hashed_Maps A.18.5(37/5)
  in Ada.Containers.Hashed_Sets A.18.8(49/5)
  in Ada.Containers.Multiway_Trees A.18.10(42/5), A.18.10(44/5)
  in Ada.Containers.Ordered_Maps A.18.6(50/5), A.18.6(51.1/5), A.18.6(51.2/5)
  in Ada.Containers.Ordered_Sets A.18.9(60/5)
  in Ada.Containers.Vectors A.18.2(73/5), A.18.2(74.1/5), A.18.2(74.2/5)
  in Ada.Environment_Variables A.17(8/5)
Iterate_Children
  in Ada.Containers.Multiway_Trees A.18.10(68.1/5), A.18.10(68/5), A.18.10(70/5)
Iterate_Subtree
  in Ada.Containers.Multiway_Trees A.18.10(43.1/5), A.18.10(43/5), A.18.10(45.1/5), A.18.10(45/5)
iterated_component_association 4.3.3(5.1/5)
  used 4.3.3(5/5), P.1
iterated_element_association 4.3.5(21/5)
  used 4.3.5(18/5), 4.5.10(3/5), P.1
iterating procedure 5.5.3(9/5), 5.5.3(27/5)
iteration cursor subtype 5.5.1(6/5)
iteration_scheme 5.5(3/5)
  used 5.5(2), P.1
iterator 1.3.3(16/5)
  array component 5.5.2(3/3)

container element 5.5.2(3/3)
forward 5.5.2(4/5)
generalized 5.5.2(3/3)
parallel 5.5.2(4/5)
reverse 5.5.2(4/5)
sequential 5.5.2(4/5)
iterator construct 5.5(6.2/5)
iterator filter 1.3.3(18/5), 5.5(6.2/5)
iterator object 5.5.1(6/5)
iterator type 5.5.1(6/5)
iterator_actual_parameter_part 5.5.3(5/5)
used 5.5.3(4/5), P.1
Iterator_Element aspect 5.5.1(9/5)
iterator_filter 5.5(4.1/5)
used 5.5(4/5), 5.5.2(2/5), 5.5.3(2/5), P.1
Iterator_Interfaces
child of Ada 5.5.1(2/5)
iterator_parameter_association 5.5.3(6/5)
used 5.5.3(5/5), P.1
iterator_parameter_specification 5.5.3(3/5)
used 5.5.3(2/5), P.1
iterator_procedure_call 5.5.3(4/5)
used 5.5.3(2/5), P.1
iterator_specification 5.5.2(2/5)
used 4.3.3(5.1/5), 4.3.5(21/5), 4.5.8(1/3), 5.5(3/5), P.1
Iterator_View aspect 5.5.1(9.1/5)

J

j
in Ada.Numerics.Generic_Complex_-Types G.1.1(5)
in Interfaces.Fortran B.5(10)
Jorvik D.13(1/5)

K

Key
in Ada.Containers.Hashed_Maps A.18.5(13.1/5), A.18.5(13/5)
in Ada.Containers.Hashed_Sets A.18.8(51.1/5), A.18.8(51/5)
in Ada.Containers.Ordered_Maps A.18.6(12.1/5), A.18.6(12/5)
in Ada.Containers.Ordered_Sets A.18.9(64.1/5), A.18.9(64/5)
key type
container aggregate 4.3.5(8/5), 4.3.5(9/5)
key_choice 4.3.5(20/5)
used 4.3.5(19/5), P.1
key_choice_list 4.3.5(19/5)
used 4.3.5(18/5), P.1
keyword 2.9(2.c/5)
Kind
in Ada.Directories A.16(25/2), A.16(40/2)
known discriminants 3.7(26)
known on entry
postcondition 6.1.1(20/5)
known to be constrained 3.3(23.1/3)
known to denote the same object 6.4.1(6.5/3)
known to have no variable views 3.3(13.1/5)
known to refer to the same object 6.4.1(6.12/3)
Known_Conflict_Checks conflict check policy 9.10.1(22/5)
known_discriminant_part 3.7(4)
used 3.2.1(3/3), 3.7(2), 9.1(2/3), 9.4(2/3), P.1
Known_Parallel_Conflict_Checks conflict check policy 9.10.1(12/5)
Known_Tasking_Conflict_Checks conflict check policy 9.10.1(16/5)

L

label 5.1(7)
used 5.1(2/3), 5.1(3), P.1
Landau symbol O(X) A.18(3/2)
language
interface to assembly C.1(4/3)
interface to non-Ada B(1)
in Ada.Locales A.19(6/3)
Language code standard 1.2(1.1/3)
language-defined categories [*partial*] 3.2(10/2)
language-defined category
of types 3.2(2/5)
language-defined check 11.5(2/3), 11.6(1/3)
language-defined class [*partial*] 3.2(10/2)
of types 3.2(2/5)
Language-defined constants Q.5(1/3)
Language-defined exceptions Q.4(1/3)
Language-Defined Library Units A(1)
Language-defined objects Q.5(1/3)
Language-defined packages Q.1(1/3)
Language-defined subprograms Q.3(1/3)
Language-defined subtypes Q.2(1/3)
Language-defined types Q.2(1/3)
Language-defined values Q.5(1/3)
Language_Code
in Ada.Locales A.19(4/4)
Language_Unknown
in Ada.Locales A.19(5/3)
Last
in Ada.Containers.Doubly_Linked_-Lists A.18.3(35/5)
in Ada.Containers.Ordered_Maps A.18.6(31/5)
in Ada.Containers.Ordered_Sets A.18.9(43/5)
in Ada.Containers.Vectors A.18.2(61/5)
in Ada.Iterator_Interfaces 5.5.1(4/3)
Last attribute 3.5(13), 3.6.2(5)
last element
of a hashed set A.18.8(68/2)
of a set A.18.7(6/2)
of an ordered set A.18.9(81/3)
last node
of a hashed map A.18.5(46/2)
of a map A.18.4(6/2)
of an ordered map A.18.6(58/3)
Last(N) attribute 3.6.2(6)
last_bit 13.5.1(6)
used 13.5.1(3), P.1
Last_Bit attribute 13.5.2(4/2)
Last_Child
in Ada.Containers.Multiway_Trees A.18.10(62.1/5), A.18.10(62/5)
Last_Child_Element
in Ada.Containers.Multiway_Trees A.18.10(63.1/5), A.18.10(63/5)
Last_Element
in Ada.Containers.Doubly_Linked_-Lists A.18.3(36/5)
in Ada.Containers.Ordered_Maps A.18.6(32/5)
in Ada.Containers.Ordered_Sets A.18.9(44/5)
in Ada.Containers.Vectors A.18.2(62/5)
Last_Index
in Ada.Containers.Vectors A.18.2(60/5)
Last_Key
in Ada.Containers.Ordered_Maps A.18.6(33/5)
Last_Valid attribute 3.5.5(7.3/4)
lateness D.9(12/5)
Latin-1 3.5.2(2/5)
Latin_1
child of Ada.Characters A.3.3(3/5)
Layout aspect 13.5(1)
Layout_Error
in Ada.IO_Exceptions A.13(4)
in Ada.Text_IO A.10.1(85/5)
LC_A
in Ada.Characters.Latin_1 A.3.3(13)
LC_A_Acute
in Ada.Characters.Latin_1 A.3.3(25)
LC_A_Circumflex
in Ada.Characters.Latin_1 A.3.3(25)
LC_A_Diaeresis
in Ada.Characters.Latin_1 A.3.3(25)
LC_A_Grave
in Ada.Characters.Latin_1 A.3.3(25)
LC_A_Ring
in Ada.Characters.Latin_1 A.3.3(25)
LC_A_Tilde
in Ada.Characters.Latin_1 A.3.3(25)
LC_AE_Diphthong
in Ada.Characters.Latin_1 A.3.3(25)
LC_B
in Ada.Characters.Latin_1 A.3.3(13)
LC_C
in Ada.Characters.Latin_1 A.3.3(13)
LC_C_Cedilla
in Ada.Characters.Latin_1 A.3.3(25)
LC_D
in Ada.Characters.Latin_1 A.3.3(13)
LC_E
in Ada.Characters.Latin_1 A.3.3(13)

LC_E_Acute
  in Ada.Characters.Latin_1 A.3.3(25)
LC_E_Circumflex
  in Ada.Characters.Latin_1 A.3.3(25)
LC_E_Diaeresis
  in Ada.Characters.Latin_1 A.3.3(25)
LC_E_Grave
  in Ada.Characters.Latin_1 A.3.3(25)
LC_F
  in Ada.Characters.Latin_1 A.3.3(13)
LC_G
  in Ada.Characters.Latin_1 A.3.3(13)
LC_German_Sharp_S
  in Ada.Characters.Latin_1 A.3.3(24)
LC_H
  in Ada.Characters.Latin_1 A.3.3(13)
LC_I
  in Ada.Characters.Latin_1 A.3.3(13)
LC_I_Acute
  in Ada.Characters.Latin_1 A.3.3(25)
LC_I_Circumflex
  in Ada.Characters.Latin_1 A.3.3(25)
LC_I_Diaeresis
  in Ada.Characters.Latin_1 A.3.3(25)
LC_I_Grave
  in Ada.Characters.Latin_1 A.3.3(25)
LC_Icelandic_Eth
  in Ada.Characters.Latin_1 A.3.3(26)
LC_Icelandic_Thorn
  in Ada.Characters.Latin_1 A.3.3(26)
LC_J
  in Ada.Characters.Latin_1 A.3.3(13)
LC_K
  in Ada.Characters.Latin_1 A.3.3(13)
LC_L
  in Ada.Characters.Latin_1 A.3.3(13)
LC_M
  in Ada.Characters.Latin_1 A.3.3(13)
LC_N
  in Ada.Characters.Latin_1 A.3.3(13)
LC_N_Tilde
  in Ada.Characters.Latin_1 A.3.3(26)
LC_O
  in Ada.Characters.Latin_1 A.3.3(13)
LC_O_Acute
  in Ada.Characters.Latin_1 A.3.3(26)
LC_O_Circumflex
  in Ada.Characters.Latin_1 A.3.3(26)
LC_O_Diaeresis
  in Ada.Characters.Latin_1 A.3.3(26)
LC_O_Grave
  in Ada.Characters.Latin_1 A.3.3(26)
LC_O_Oblique_Stroke
  in Ada.Characters.Latin_1 A.3.3(26)
LC_O_Tilde
  in Ada.Characters.Latin_1 A.3.3(26)
LC_P
  in Ada.Characters.Latin_1 A.3.3(14)
LC_Q
  in Ada.Characters.Latin_1 A.3.3(14)
LC_R
  in Ada.Characters.Latin_1 A.3.3(14)
LC_S
  in Ada.Characters.Latin_1 A.3.3(14)
LC_T
  in Ada.Characters.Latin_1 A.3.3(14)
LC_U
  in Ada.Characters.Latin_1 A.3.3(14)
LC_U_Acute
  in Ada.Characters.Latin_1 A.3.3(26)
LC_U_Circumflex
  in Ada.Characters.Latin_1 A.3.3(26)
LC_U_Diaeresis
  in Ada.Characters.Latin_1 A.3.3(26)
LC_U_Grave
  in Ada.Characters.Latin_1 A.3.3(26)
LC_V
  in Ada.Characters.Latin_1 A.3.3(14)
LC_W
  in Ada.Characters.Latin_1 A.3.3(14)
LC_X
  in Ada.Characters.Latin_1 A.3.3(14)
LC_Y
  in Ada.Characters.Latin_1 A.3.3(14)
LC_Y_Acute
  in Ada.Characters.Latin_1 A.3.3(26)
LC_Y_Diaeresis
  in Ada.Characters.Latin_1 A.3.3(26)
LC_Z
  in Ada.Characters.Latin_1 A.3.3(14)
Leading_Nonseparate
  in Interfaces.COBOL B.4(23)
Leading_Part attribute A.5.3(54)
Leading_Separate
  in Interfaces.COBOL B.4(23)
leaf node
  of a tree A.18.10(4/3)
Leap_Seconds_Count *subtype of* Integer
  in Ada.Calendar.Arithmetic 9.6.1(11/2)
leaving 7.6.1(3/5)
left 7.6.1(3/5)
left parenthesis 2.1(15/5)
left square bracket 2.1(15/5)
Left_Angle_Quotation
  in Ada.Characters.Latin_1 A.3.3(21/3)
Left_Curly_Bracket
  in Ada.Characters.Latin_1 A.3.3(14)
Left_Parenthesis
  in Ada.Characters.Latin_1 A.3.3(8)
Left_Square_Bracket
  in Ada.Characters.Latin_1 A.3.3(12)
legal
  construct 1.1.2(27)
  partition 1.1.2(29)
legality determinable via semantic dependences 10(3.c)
legality rules 1.1.2(27)
length
  of a dimension of an array 3.6(13)
  of a list container A.18.3(3/2)
  of a map A.18.4(5/2)
  of a one-dimensional array 3.6(13)
  of a set A.18.7(5/2)
  of a vector container A.18.2(2/2)
  in Ada.Containers.Doubly_Linked_Lists A.18.3(11/5)
  in Ada.Containers.Hashed_Maps A.18.5(10/5)
  in Ada.Containers.Hashed_Sets A.18.8(12/5)
  in Ada.Containers.Ordered_Maps A.18.6(9/5)
  in Ada.Containers.Ordered_Sets A.18.9(11/5)
  in Ada.Containers.Vectors A.18.2(21/5)
  in Ada.Strings.Bounded A.4.4(9)
  in Ada.Strings.Unbounded A.4.5(6)
  in Ada.Text_IO.Editing F.3.3(11)
  in Interfaces.COBOL B.4(34), B.4(39), B.4(44)
Length attribute 3.6.2(9)
Length(N) attribute 3.6.2(10)
Length_Check 11.5(15)
  [*partial*] 4.5.1(8), 4.6(37), 4.6(52)
Length_Error
  in Ada.Strings A.4.1(5)
Length_Range *subtype of* Natural
  in Ada.Strings.Bounded A.4.4(8)
less than operator 4.5(1), 4.5.2(1)
less than or equal operator 4.5(1), 4.5.2(1)
less-than sign 2.1(15/5)
Less_Case_Insensitive
  child of Ada.Strings A.4.10(13/5)
  child of Ada.Strings.Bounded A.4.10(18/5)
  child of Ada.Strings.Fixed A.4.10(16/3)
  child of Ada.Strings.Unbounded A.4.10(21/5)
Less_Than_Sign
  in Ada.Characters.Latin_1 A.3.3(10)
letter
  a category of Character A.3.2(24)
letter_lowercase 2.1(9/2)
  used 2.3(3/2), P.1
letter_modifier 2.1(9.2/2)
  used 2.3(3/2), P.1
letter_other 2.1(9.3/2)
  used 2.3(3/2), P.1
Letter_Set
  in Ada.Strings.Maps.Constants A.4.6(4)
letter_titlecase 2.1(9.1/2)
  used 2.3(3/2), P.1
letter_uppercase 2.1(8/2)
  used 2.3(3/2), P.1
level
  accessibility 3.10.2(3/5)
  library 3.10.2(22)
lexical element 2.2(1)
lexicographic order 4.5.2(26/3)
LF
  in Ada.Characters.Latin_1 A.3.3(5)
library 10.1.4(9/5)
  [*partial*] 10.1.1(9)
  informal introduction 10(2)
  See also library level, library unit, library_item
library level 3.10.2(22)
library unit 1.3.3(19/5), 10.1(3), 10.1.1(9)
  informal introduction 10(2)

See also language-defined library units
library unit aspect 13.1.1(32/5)
library unit pragma J.15(7/5)
All_Calls_Remote J.15.15(7/5)
Elaborate_Body J.15.14(7/5)
Preelaborate J.15.14(7/5)
Pure J.15.14(7/5)
Remote_Call_Interface J.15.15(7/5)
Remote_Types J.15.15(7/5)
Shared_Passive J.15.15(7/5)
library_item 10.1.1(4)
informal introduction 10(2)
used 10.1.1(3), P.1
library_unit_body 10.1.1(7)
used 10.1.1(4), P.1
library_unit_declaration 10.1.1(5)
used 10.1.1(4), P.1
library_unit_renaming_declaration 10.1.1(6)
used 10.1.1(4), P.1
lifetime 3.10.2(3/5)
limited interface 3.9.4(5/2)
limited type 1.3.1(46/5), 7.5(3/3)
becoming nonlimited 7.3.1(5/1), 7.5(16)
immutably 7.5(8.1/3)
limited view 10.1.1(12.1/2)
Limited_Controlled
in Ada.Finalization 7.6(7/5)
limited_with_clause 10.1.2(4.1/2)
used 10.1.2(4/2), P.1
line 2.2(2/3)
in Ada.Text_IO A.10.1(38/5)
line terminator A.10(7)
Line_Length
in Ada.Text_IO A.10.1(25/5)
link name B.1(35)
link-time error
See post-compilation error 1.1.2(29)
See post-compilation error 1.1.5(4)
Link_Name aspect B.1(1/3)
Linker_Options pragma B.1(8), L(19)
linking
See partition building 10.2(2/5)
List
in Ada.Containers.Doubly_Linked_Lists A.18.3(6/5), A.18.3(51.2/5)
list container A.18.3(1/2)
List pragma 2.8(21), L(20)
List_Iterator_Interfaces
in Ada.Containers.Doubly_Linked_Lists A.18.3(9.3/3), A.18.3(51.7/5)
literal 4.2(1)
based 2.4.2(1)
decimal 2.4.1(1)
numeric 2.4(1)
See also aggregate 4.3(1)
little endian 13.5.3(2)
load time C.4(3)
local to 8.1(14)
Local_Image
in Ada.Calendar.Formatting 9.6.1(35.1/5)
local_name 13.1(3)
used 13.3(2), 13.4(2), 13.5.1(2/5), 13.5.1(3), C.5(3), J.15.2(2/5), J.15.3(2/3), J.15.5(2/3), J.15.5(3/3), J.15.5(4/3), J.15.6(2/3), J.15.8(2/3), J.15.8(3/3), J.15.8(4/3), J.15.8(5/3), J.15.8(6/3), J.15.8(7/3), J.15.13(2/3), L(3.1/3), L(4.1/3), L(5.1/3), L(8.1/3), L(9), L(13.1/3), L(14.1/3), L(14.2/3), L(14.3/3), L(21.2/3), L(24.1/3), L(37.2/3), L(38.1/3), L(39.1/3), P.1
Local_Time_Offset
in Ada.Calendar.Time_Zones 9.6.1(6/5)
locale A.19(1/3)
active A.19(8/3)
Locales
child of Ada A.19(3/5)
lock-free C.6.1(5/5)
locking policy D.3(6/2)
Ceiling_Locking D.3(7)
Locking_Policy pragma D.3(3), L(21)
Log
in Ada.Numerics.Generic_Complex_Elementary_Functions G.1.2(3)
in Ada.Numerics.Generic_Elementary_Functions A.5.1(4)
Logical
in Interfaces.Fortran B.5(7)
logical operator 4.5.1(2)
See also not operator 4.5.6(3)
logical thread of control 1.3.4(9/5), 9(1/5)
logical_operator 4.5(2)
long
in Interfaces.C B.3(7)
Long_Binary
in Interfaces.COBOL B.4(10)
long_double
in Interfaces.C B.3(17)
Long_Float 3.5.7(15), 3.5.7(16/5), 3.5.7(17)
Long_Floating
in Interfaces.COBOL B.4(9)
Long_Integer 3.5.4(22), 3.5.4(25), 3.5.4(28)
Look_Ahead
in Ada.Text_IO A.10.1(43/5)
loop
parallel 5.5(8.1/5)
sequential 5.5(9/5)
loop body procedure 5.5.3(15/5)
loop cursor 5.5.2(12/5)
loop iterator 5.5.2(10/5)
container element iterator 5.5.2(12/5)
loop parameter 5.5(6/5), 5.5.2(7/5)
loop_parameter_specification 5.5(4/5)
used 4.3.5(21/5), 4.5.8(1/3), 5.5(3/5), P.1
loop_parameter_subtype_indication 5.5.2(2.1/5)
used 5.5.2(2/5), P.1
loop_statement 5.5(2)
used 5.1(5/5), P.1
low line 2.1(15/5)
low-level programming C(1)
Low_Line
in Ada.Characters.Latin_1 A.3.3(12)
Low_Order_First 13.5.3(2)
in Interfaces.COBOL B.4(25)
in System 13.7(15/2)
lower bound
of a range 3.5(4)
lower-case letter
a category of Character A.3.2(25)
Lower_Case_Map
in Ada.Strings.Maps.Constants A.4.6(5)
Lower_Set
in Ada.Strings.Maps.Constants A.4.6(4)
LR(1) 1.1.4(14.a)

M

Machine attribute A.5.3(60)
machine code insertion 13.8(1), C.1(2/5)
machine numbers
of a fixed point type 3.5.9(8/2)
of a floating point type 3.5.7(8)
machine scalar 13.3(8.1/3)
Machine_Code
child of System 13.8(7)
Machine_Emax attribute A.5.3(8)
Machine_Emin attribute A.5.3(7)
Machine_Mantissa attribute A.5.3(6)
Machine_Overflows attribute A.5.3(12), A.5.4(4)
Machine_Radix aspect F.1(1/5)
Machine_Radix attribute A.5.3(2), A.5.4(2)
Machine_Radix clause 13.3(7/2), F.1(1/5)
Machine_Rounding attribute A.5.3(41.1/2)
Machine_Rounds attribute A.5.3(11), A.5.4(3)
macro
See generic unit 12(1)
Macron
in Ada.Characters.Latin_1 A.3.3(21/3)
main subprogram
for a partition 10.2(7)
malloc
See allocator 4.8(1)
Map
in Ada.Containers.Hashed_Maps A.18.5(3/5), A.18.5(37.3/5)
in Ada.Containers.Ordered_Maps A.18.6(4/5), A.18.6(51.4/5)
map container A.18.4(1/2)
Map_Iterator_Interfaces
in Ada.Containers.Hashed_Maps A.18.5(6.3/5), A.18.5(37.8/5)
in Ada.Containers.Ordered_Maps A.18.6(7.3/5), A.18.6(51.9/5)
Maps
child of Ada.Strings A.4.2(3/5)
mark_non_spacing 2.1(9.4/2), 2.1(9.5/2)

used 2.3(3.1/3), P.1
mark_spacing_combining
used 2.3(3.1/3), P.1
marshalling E.4(9)
Masculine_Ordinal_Indicator
in Ada.Characters.Latin_1 A.3.3(22)
master 1.3.4(10/5), 7.6.1(3/5)
master construct 1.3.3(21/5), 7.6.1(3/5)
master of a call 3.10.2(10.1/3)
match
a character to a pattern character A.4.2(54)
a character to a pattern character, with respect to a character mapping function A.4.2(64)
a string to a pattern string A.4.2(54)
value of nonoverridable aspect 13.1.1(18.4/5)
matching components 4.5.2(16)
Max
in Ada.Numerics.Big_Numbers.Big_Integers A.5.6(18/5)
in Ada.Numerics.Big_Numbers.Big_Reals A.5.7(21/5)
Max attribute 3.5(19)
Max_Alignment_For_Allocation attribute 13.11.1(4/5)
Max_Asynchronous_Select_Nesting restriction D.7(18/1)
Max_Base_Digits 3.5.7(6)
in System 13.7(8)
Max_Binary_Modulus 3.5.4(7)
in System 13.7(7)
Max_Decimal_Digits
in Ada.Decimal F.2(5)
Max_Delta
in Ada.Decimal F.2(4)
Max_Digits 3.5.7(6)
in System 13.7(8)
Max_Digits_Binary
in Interfaces.COBOL B.4(11)
Max_Digits_Long_Binary
in Interfaces.COBOL B.4(11)
Max_Entry_Queue_Length aspect D.4(18/5)
Max_Entry_Queue_Length restriction D.7(19.1/2)
Max_Image_Length restriction H.4(23.11/5)
Max_Image_Width
in Ada.Numerics.Discrete_Random A.5.2(25)
in Ada.Numerics.Float_Random A.5.2(13)
Max_Int 3.5.4(14)
in System 13.7(6)
Max_Length
in Ada.Strings.Bounded A.4.4(5)
Max_Mantissa
in System 13.7(9)
Max_Nonbinary_Modulus 3.5.4(7)
in System 13.7(7)
Max_Picture_Length
in Ada.Text_IO.Editing F.3.3(8)
Max_Protected_Entries restriction D.7(14)
Max_Scale
in Ada.Decimal F.2(3)
Max_Select_Alternatives restriction D.7(12)
Max_Size_In_Storage_Elements attribute 13.11.1(3/5)
Max_Storage_At_Blocking restriction D.7(17/1)
Max_Task_Entries restriction D.7(13)
Max_Tasks restriction D.7(19/1)
maximum box error
for a component of the result of evaluating a complex function G.2.6(3/5)
maximum line length A.10(11)
maximum page length A.10(11)
maximum relative error
for a component of the result of evaluating a complex function G.2.6(3/5)
for the evaluation of an elementary function G.2.4(2)
Maximum_Length
in Ada.Containers.Vectors A.18.2(12.3/5)
Meaningful_For
in Ada.Containers.Multiway_Trees A.18.10(22.1/5)
Members
in Ada.Execution_Time.Group_Budgets D.14.2(8/2)
Membership
in Ada.Strings A.4.1(6)
membership test 4.5.2(2/3)
membership_choice 4.4(3.2/4)
used 4.4(3.1/5), P.1
membership_choice_list 4.4(3.1/5)
used 4.4(3/4), P.1
Memory_Size
in System 13.7(13)
mentioned
in a with_clause 10.1.2(6/2)
Merge
in Ada.Containers.Doubly_Linked_Lists A.18.3(50/5)
in Ada.Containers.Vectors A.18.2(78/5)
message
See dispatching call 3.9.2(1/5)
method
See dispatching subprogram 3.9.2(1/5)
methodological restriction 10.1.3(13.a)
metrics 1.1.2(35)
Micro_Sign
in Ada.Characters.Latin_1 A.3.3(22)
Microseconds
in Ada.Real_Time D.8(14/2)
Middle_Dot
in Ada.Characters.Latin_1 A.3.3(22)
Milliseconds
in Ada.Real_Time D.8(14/2)
Min
in Ada.Numerics.Big_Numbers.Big_Integers A.5.6(18/5)
in Ada.Numerics.Big_Numbers.Big_Reals A.5.7(21/5)
Min attribute 3.5(16)
Min_Delta
in Ada.Decimal F.2(4)
Min_Handler_Ceiling
in Ada.Execution_Time.Group_Budgets D.14.2(7/2)
in Ada.Execution_Time.Timers D.14.1(6/2)
Min_Int 3.5.4(14)
in System 13.7(6)
Min_Scale
in Ada.Decimal F.2(3)
minus 2.1(15/5)
minus operator 4.5(1), 4.5.3(1), 4.5.4(1)
Minus_Sign
in Ada.Characters.Latin_1 A.3.3(8)
Minute
in Ada.Calendar.Formatting 9.6.1(25/2)
Minute_Number *subtype of* Natural
in Ada.Calendar.Formatting 9.6.1(20/2)
Minutes
in Ada.Real_Time D.8(14/2)
mixed-language programs B(1), C.1(4/3)
Mod attribute 3.5.4(16.1/2)
mod operator 4.5(1), 4.5.5(1)
mod_clause J.8(1)
used 13.5.1(2/5), P.1
mode 6.1(16)
used 6.1(15/5), 12.4(2/3), P.1
in Ada.Direct_IO A.8.4(9)
in Ada.Sequential_IO A.8.1(9)
in Ada.Streams.Stream_IO A.12.1(11)
in Ada.Text_IO A.10.1(12)
mode conformance 6.3.1(16/3)
required 8.5.4(4/5), 8.5.4(5/3), 12.6(7/3), 12.6(8/3), 13.1(13.2/5)
mode of operation
nonstandard 1.1.5(11)
standard 1.1.5(11)
Mode_Error
in Ada.Direct_IO A.8.4(18)
in Ada.IO_Exceptions A.13(4)
in Ada.Sequential_IO A.8.1(15)
in Ada.Streams.Stream_IO A.12.1(26)
in Ada.Text_IO A.10.1(85/5)
Model attribute A.5.3(68), G.2.2(7)
model interval G.2.1(4)
associated with a value G.2.1(4)
model number G.2.1(3)
model-oriented attributes
of a floating point subtype A.5.3(63)
Model_Emin attribute A.5.3(65), G.2.2(4)
Model_Epsilon attribute A.5.3(66)

Model_Mantissa attribute A.5.3(64), G.2.2(3/2)
Model_Small attribute A.5.3(67)
Modification_Time
  in Ada.Directories A.16(27/2), A.16(42/2)
modular type 3.5.4(1)
Modular_Arithmetic
  child of System.Atomic_Operations C.6.5(3/5)
Modular_IO
  in Ada.Text_IO A.10.1(57)
modular_type_definition 3.5.4(4)
  used 3.5.4(2), P.1
module
  See package 7(1)
modulus
  of a modular type 3.5.4(7)
  in Ada.Numerics.Generic_Complex_-Arrays G.3.2(10/2), G.3.2(30/2)
  in Ada.Numerics.Generic_Complex_-Types G.1.1(9)
Modulus attribute 3.5.4(17)
Monday
  in Ada.Calendar.Formatting 9.6.1(17/2)
Month
  in Ada.Calendar 9.6(13)
  in Ada.Calendar.Formatting 9.6.1(22/2)
Month_Number *subtype of* Integer
  in Ada.Calendar 9.6(11/2)
More_Entries
  in Ada.Directories A.16(34/2)
Move
  in Ada.Containers.Doubly_Linked_-Lists A.18.3(18/5)
  in Ada.Containers.Hashed_Maps A.18.5(18/5)
  in Ada.Containers.Hashed_Sets A.18.8(18/5)
  in Ada.Containers.Indefinite_Holders A.18.18(22/5)
  in Ada.Containers.Multiway_Trees A.18.10(34/5)
  in Ada.Containers.Ordered_Maps A.18.6(17/5)
  in Ada.Containers.Ordered_Sets A.18.9(17/5)
  in Ada.Containers.Vectors A.18.2(35/5)
  in Ada.Strings.Fixed A.4.3(7)
multi-dimensional array 3.6(12)
Multiplication_Sign
  in Ada.Characters.Latin_1 A.3.3(24)
multiply 2.1(15/5)
multiply operator 4.5(1), 4.5.5(1)
multiplying operator 4.5.5(1)
multiplying_operator 4.5(6)
  used 4.4(5), P.1
Multiprocessors
  child of System D.16(3/5)
Multiway_Trees
  child of Ada.Containers A.18.10(7/5)
mutable 3.7(28.b)
mutates 7.6(17.6/3)
MW
  in Ada.Characters.Latin_1 A.3.3(18)

N

n-dimensional array_aggregate 4.3.3(6)
NAK
  in Ada.Characters.Latin_1 A.3.3(6)
name 4.1(2/5)
  [*partial*] 3.1(1/5)
  of (a view of) an entity 3.1(8)
  of a pragma 2.8(9)
  of an external file A.7(1)
  used 2.8(3/3), 3.2.2(4), 4.1(4), 4.1(5), 4.1(6), 4.1.5(4/3), 4.4(7/5), 4.5.10(5/5), 4.6(2), 4.8(2.1/3), 5.2(2), 5.5.2(2/5), 5.5.3(4/5), 5.7(2), 5.8(2), 6.1.2(8/5), 6.4(2), 6.4(3), 6.4(6), 8.4(3), 8.5.1(2/5), 8.5.2(2/3), 8.5.3(2/3), 8.5.4(2/3), 8.5.5(2/3), 9.5.3(2), 9.5.4(2/3), 9.8(2), 10.1.1(8), 10.1.2(4.1/2), 10.1.2(4.2/2), 10.2.1(20), 10.2.1(21), 11.2(5), 11.3(2.1/4), 11.3(2/2), 12.3(2/3), 12.3(5), 12.6(4), 12.7(2/3), 13.1(3), 13.1.1(4/5), 13.3(2), 13.11.3(3.1/4), 13.12(4.1/2), H.3.2(3), H.7.1(6/5), H.7.1(10/5), J.10(3/2), J.15.1(2/3), J.15.7(2/3), J.15.7(4/3), J.15.14(3/5), J.15.14(5/5), J.15.14(6/5), J.15.15(3/5), J.15.15(4/5), J.15.15(5/5), J.15.15(6/5), L(2.1/5), L(6.1/3), L(10), L(11), L(12.1/5), L(15.1/3), L(16), L(17.1/3), L(26.1/5), L(28.1/5), L(30.1/5), L(31.1/5), L(34.1/5), P.1
  in Ada.Direct_IO A.8.4(9), A.8.4(18.12/5), A.8.4(18.5/5)
  in Ada.Sequential_IO A.8.1(9), A.8.1(15.12/5), A.8.1(15.5/5)
  in Ada.Streams.Stream_IO A.12.1(11), A.12.1(26.12/5), A.12.1(26.5/5)
  in Ada.Text_IO A.10.1(12), A.10.1(85.12/5), A.10.1(85.5/5)
  in System 13.7(4)
name resolution rules 1.1.2(26/3)
Name_Case_Equivalence
  in Ada.Directories A.16(20.2/3)
Name_Case_Kind
  in Ada.Directories A.16(20.1/3)
Name_Error
  in Ada.Direct_IO A.8.4(18)
  in Ada.Directories A.16(43/2)
  in Ada.IO_Exceptions A.13(4)
  in Ada.Sequential_IO A.8.1(15)
  in Ada.Streams.Stream_IO A.12.1(26)
  in Ada.Text_IO A.10.1(85/5)
named
  in a use clause 8.4(7.1/2)
  in a with_clause 10.1.2(6/2)
named association 6.4(7/5), 6.4.1(2/3), 12.3(6)
named component association 4.3.1(6)
named discriminant association 3.7.1(4)
named entry index 9.5.2(21)
named number 3.3(24)
named parameter association 6.4.1(2/3)
named type 3.2.1(7/2)
named_array_aggregate 4.3.3(4/5)
  used 4.3.3(2/5), P.1
named_container_aggregate 4.3.5(16/5)
  used 4.3.5(13/5), P.1
Names
  child of Ada.Interrupts C.3.2(12/5)
Nanoseconds
  in Ada.Real_Time D.8(14/2)
Native_Binary
  in Interfaces.COBOL B.4(25)
Natural 3.5.4(12)
Natural *subtype of* Integer
  in Standard A.1(13)
NBH
  in Ada.Characters.Latin_1 A.3.3(17)
NBSP
  in Ada.Characters.Latin_1 A.3.3(21/3)
needed
  of a compilation unit by another 10.2(2/5)
  remote call interface E.2.3(18)
  shared passive library unit E.2.1(11)
needed compilation unit 1.3.3(23/5), 10.2(2/5)
needed component 1.3.1(48/5)
  extension_aggregate record_component_association_list 4.3.2(6)
  record_aggregate record_component_association_list 4.3.1(9/5)
needs finalization 7.6(9.1/2)
  language-defined type A.4.5(72.1/2), A.5.2(15.1/2), A.5.2(27.1/2), A.5.6(23/5), A.5.7(26/5), A.8.1(17/2), A.8.4(20/2), A.10.1(86/2), A.12.1(27.1/2), A.16(102/2), A.18.2(147.3/3), A.18.2(84/2), A.18.3(56/2), A.18.3(86.3/3), A.18.4(4/2), A.18.4(41.3/3), A.18.7(4/2), A.18.7(36.4/3), A.18.7(96.2/3), A.18.10(124/3), A.18.10(73/3), A.18.18(27/3), A.18.18(54/3), D.14.2(13/2), D.15(8/2)
NEL
  in Ada.Characters.Latin_1 A.3.3(17)
new
  See allocator 4.8(1)
New_Char_Array
  in Interfaces.C.Strings B.3.1(9)
New_Line
  in Ada.Strings.Text_Buffers A.4.12(12/5)
  in Ada.Text_IO A.10.1(28/5)
New_Line_Count
  in Ada.Strings.Text_Buffers A.4.12(5/5)
New_Page
  in Ada.Text_IO A.10.1(31/5)
New_String
  in Interfaces.C.Strings B.3.1(10)

New_Vector
  in Ada.Containers.Vectors A.18.2(14.1/5)
newly constructed 4.4(9.7/5)
Next
  in Ada.Containers.Doubly_Linked_-Lists A.18.3(37.1/5), A.18.3(37/5), A.18.3(39.1/5), A.18.3(39/5)
  in Ada.Containers.Hashed_Maps A.18.5(28.1/5), A.18.5(28/5), A.18.5(29.1/5), A.18.5(29/5)
  in Ada.Containers.Hashed_Sets A.18.8(41.1/5), A.18.8(41/5), A.18.8(42.1/5), A.18.8(42/5)
  in Ada.Containers.Ordered_Maps A.18.6(34.1/5), A.18.6(34/5), A.18.6(35.1/5), A.18.6(35/5)
  in Ada.Containers.Ordered_Sets A.18.9(45.1/5), A.18.9(45/5), A.18.9(46.1/5), A.18.9(46/5)
  in Ada.Containers.Vectors A.18.2(63.1/5), A.18.2(63/5), A.18.2(64.1/5), A.18.2(64/5)
  in Ada.Iterator_Interfaces 5.5.1(3/3), 5.5.1(4.7/5)
Next_Sibling
  in Ada.Containers.Multiway_Trees A.18.10(64.1/5), A.18.10(64/5), A.18.10(65.1/5), A.18.10(65/5)
No_Abort_Statements restriction D.7(5/3)
No_Access_Parameter_Allocators restriction H.4(8.3/3)
No_Access_Subprograms restriction H.4(17)
No_Allocators restriction H.4(7)
No_Anonymous_Allocators restriction H.4(8.1/3)
No_Break_Space
  in Ada.Characters.Latin_1 A.3.3(21/3)
No_Coextensions restriction H.4(8.2/3)
No_Conflict_Checks conflict check policy 9.10.1(22/5)
No_Controlled_Parts aspect H.4.1(2/5)
No_Delay restriction H.4(21)
No_Dependence restriction 13.12.1(6/2)
No_Deposit aspect 6.5.1(1.a/5)
No_Dispatch restriction H.4(19)
No_Dynamic_Attachment restriction D.7(10/3)
No_Dynamic_CPU_Assignment restriction D.7(10.1/4)
No_Dynamic_Priorities restriction D.7(9/2)
No_Element
  in Ada.Containers.Doubly_Linked_-Lists A.18.3(9/2), A.18.3(51.5/5)
  in Ada.Containers.Hashed_Maps A.18.5(6/2), A.18.5(37.6/5)
  in Ada.Containers.Hashed_Sets A.18.8(6/2), A.18.8(59.5/5)
  in Ada.Containers.Multiway_Trees A.18.10(11/3), A.18.10(70.5/5)
  in Ada.Containers.Ordered_Maps A.18.6(7/2), A.18.6(51.7/5)
  in Ada.Containers.Ordered_Sets A.18.9(7/2), A.18.9(74.5/5)
  in Ada.Containers.Vectors A.18.2(11/2), A.18.2(79.5/5)
No_Exceptions restriction H.4(12/5)
No_Fixed_Point restriction H.4(15)
No_Floating_Point restriction H.4(14)
No_Hidden_Indirect_Globals restriction H.4(23.2/5)
No_Implementation_Aspect_Specifications restriction 13.12.1(1.1/3)
No_Implementation_Attributes restriction 13.12.1(2/2)
No_Implementation_Identifiers restriction 13.12.1(2.1/3)
No_Implementation_Pragmas restriction 13.12.1(3/2)
No_Implementation_Units restriction 13.12.1(3.1/3)
No_Implicit_Heap_Allocations restriction D.7(8)
No_Index
  in Ada.Containers.Vectors A.18.2(7/2)
No_IO restriction H.4(20/5)
No_Local_Allocators restriction H.4(8/1)
No_Local_Protected_Objects restriction D.7(10.2/3)
No_Local_Timing_Events restriction D.7(10.3/3)
No_Nested_Finalization restriction D.7(4/3)
No_Obsolescent_Features restriction 13.12.1(4/3)
No_Parallel_Conflict_Checks conflict check policy 9.10.1(10/5)
No_Protected_Type_Allocators restriction D.7(10.4/2)
No_Protected_Types restriction H.4(5)
No_Recursion restriction H.4(22)
No_Reentrancy restriction H.4(23)
No_Relative_Delay restriction D.7(10.6/3)
No_Requeue_Statements restriction D.7(10.7/3)
No_Return aspect 6.5.1(3.2/3)
No_Return pragma J.15.2(2/5), L(21.2/3)
No_Select_Statements restriction D.7(10.8/3)
No_Specific_Termination_Handlers restriction D.7(10.9/3)
No_Specification_of_Aspect restriction 13.12.1(6.1/3)
No_Standard_Allocators_After_Elaboration restriction D.7(19.2/3)
No_Tag
  in Ada.Tags 3.9(6.1/2)
No_Task_Allocators restriction D.7(7)
No_Task_Hierarchy restriction D.7(3/3)
No_Task_Termination restriction D.7(15.1/2)
No_Tasking_Conflict_Checks conflict check policy 9.10.1(11/5)
No_Tasks_Unassigned_To_CPU restriction D.7(10.10/4)
No_Terminate_Alternatives restriction D.7(6)
No_Unchecked_Access restriction H.4(18)
No_Unrecognized_Aspects restriction 13.12.1(6.4/5)
No_Unrecognized_Pragmas restriction 13.12.1(6.5/5)
No_Unspecified_Globals restriction H.4(23.1/5)
No_Use_Of_Attribute restriction 13.12.1(6.2/3)
No_Use_Of_Pragma restriction 13.12.1(6.3/3)
node
  of a list A.18.3(2/2)
  of a map A.18.4(5/2)
  of a tree A.18.10(2/4)
Node_Count
  in Ada.Containers.Multiway_Trees A.18.10(17/5)
nominal subtype 1.3.1(49/5), 3.3(23/5), 3.3.1(8/2)
  associated with a dereference 4.1(9/3)
  associated with a type_conversion 4.6(27)
  associated with an indexed_component 4.1.1(5)
  of a component 3.6(20)
  of a formal parameter 6.1(23/2)
  of a function result 6.1(23/2)
  of a generic formal object 12.4(9/2)
  of a name 4.1(9.b)
  of a qualified expression 4.7(3.1/3)
  of a record component 3.8(14)
  of an attribute reference 4.1.4(9/4)
  of an enumeration literal 3.5.1(6/3)
  of the result of a function_call 6.4(12/2)
nominal type 3.3(23/5)
Non_Preemptive
  child of Ada.Dispatching D.2.4(2.2/5)
Non_Preemptive_FIFO_Within_Priorities task disp. policy D.2.4(2/2)
nonblocking 9.5(21/5)
Nonblocking aspect 9.5(19/5)
nonblocking region 9.5(43/5)
nonconfirming
  aspect specification 13.1(18.2/5)
  operational aspect 13.1(18.2/5)
  operational item 13.1(18.2/5)
  representation aspect 13.1(18.2/5)
  representation item 13.1(18.2/5)
  representation value 13.1(18.2/5)
nondispatching call
  on a dispatching operation 3.9.2(1/5)
nonexistent 13.11.2(10/4), 13.11.2(16/3)
nongraphic character 4.10(12/5)
nonlimited interface 3.9.4(5/2)
nonlimited type 1.3.1(50/5), 7.5(7)
  becoming nonlimited 7.3.1(5/1), 7.5(16)

nonlimited_with_clause 10.1.2(4.2/2)
used 10.1.2(4/2), P.1
nonnormative
See informative 1.1.2(18)
nonoverridable
aspect 13.1.1(18.3/5)
nonoverridable aspect
Aggregate 4.3.5(12/5)
Constant_Indexing 4.1.6(5.1/4)
Default_Iterator 5.5.1(11.1/4)
Implicit_Dereference 4.1.5(5.1/4)
Integer_Literal 4.2.1(6/5)
Iterator_Element 5.5.1(11.1/4)
Max_Entry_Queue_Length D.4(20/5)
No_Controlled_Parts H.4.1(3/5)
Real_Literal 4.2.1(6/5)
String_Literal 4.2.1(6/5)
Variable_Indexing 4.1.6(5.1/4)
nonreturning 6.5.1(3.2/3)
nonstandard integer type 3.5.4(26/5)
nonstandard mode 1.1.5(11)
nonstandard real type 3.5.6(8/5)
normal completion 7.6.1(2/2)
normal library unit E.2(4/3)
normal state of an object 11.6(6/5), 13.9.1(4)
[*partial*] 9.8(21), A.13(17)
normal termination
of a partition 10.2(25.c)
Normalize_Scalars pragma H.1(3), L(22)
normalized exponent A.5.3(14)
normalized number A.5.3(10)
normative 1.1.2(14)
not equal operator 4.5(1), 4.5.2(1)
not in (membership test) 4.5(1), 4.5.2(2/3)
not operator 4.5(1), 4.5.6(3)
Not_A_Specific_CPU
in System.Multiprocessors D.16(4/3)
Not_Sign
in Ada.Characters.Latin_1 A.3.3(21/3)
notes 1.1.2(38)
Notwithstanding 3.10.2(21.1/5), 7.6(17.5/3), 10.1.6(6/2), 10.2(18.c/5), 13.1.1(32/5), 13.14(15.3/5), B.1(22/3), B.1(38/3), C.3.1(19/3), E.2.1(8), E.2.1(11), E.2.3(18), H.6(7/2), J.3(6)
[*partial*] J.15.5(8/3)
NUL
in Ada.Characters.Latin_1 A.3.3(5)
in Interfaces.C B.3(20/1)
null access value 4.2(9/5)
null array 3.6.1(7)
null constraint 3.2(7/2)
null extension 3.9.1(4.1/2)
null pointer
See null access value 4.2(9/5)
null procedure 6.7(3/5)
null range 3.5(4)
null record 3.8(15)
null slice 4.1.2(7)
null string literal 2.6(6)
null value
of an access type 3.10(13/2)
Null_Address
in System 13.7(12)
null_array_aggregate 4.3.3(3.1/5)
used 4.3.3(2/5), P.1
Null_Bounded_String
in Ada.Strings.Bounded A.4.4(7)
null_container_aggregate 4.3.5(14/5)
used 4.3.5(13/5), P.1
null_exclusion 3.10(5.1/2)
used 3.2.2(3/2), 3.7(5/5), 3.10(2/2), 3.10(6/2), 6.1(13/2), 6.1(15/5), 8.5.1(2/5), 12.4(2/3), P.1
Null_Id
in Ada.Exceptions 11.4.1(2/5)
Null_Occurrence
in Ada.Exceptions 11.4.1(3/5)
null_procedure_declaration 6.7(2/3)
used 3.1(3/3), 9.4(8/4), P.1
Null_Ptr
in Interfaces.C.Strings B.3.1(7)
Null_Set
in Ada.Strings.Maps A.4.2(5)
in Ada.Strings.Wide_Maps A.4.7(5)
in Ada.Strings.Wide_Wide_Maps A.4.8(5/2)
null_statement 5.1(6)
used 5.1(4/2), P.1
Null_Task_Id
in Ada.Task_Identification C.7.1(2/5)
Null_Unbounded_String
in Ada.Strings.Unbounded A.4.5(5)
Number of the Beast 6.6(6.a/3)
number sign 2.1(15/5)
Number_Base *subtype of* Integer
in Ada.Numerics.Big_Numbers.Big_Integers A.5.5(3/5)
in Ada.Text_IO A.10.1(6)
number_decimal 2.1(10/2)
used 2.3(3.1/3), P.1
number_declaration 3.3.2(2)
used 3.1(3/3), P.1
number_letter 2.1(10.1/2)
used 2.3(3/2), P.1
Number_Of_CPUs
in System.Multiprocessors D.16(5/3)
Number_Sign
in Ada.Characters.Latin_1 A.3.3(8)
numeral 2.4.1(3)
used 2.4.1(2), 2.4.1(4), 2.4.2(3), P.1
Numerator
in Ada.Numerics.Big_Numbers.Big_Reals A.5.7(7/5)
Numeric
in Interfaces.COBOL B.4(20/3)
numeric type 3.5(1)
numeric_literal 2.4(2)
used 4.4(7/5), P.1
numerics G(1)
child of Ada A.5(3/5)
Numerics_Assertion_Check 11.5(23.9/5)

O

O(f(N)) A.18(3/2)
object 1.3.1(51/5), 3.3(2)
[*partial*] 3.2(1)
required 5.2(5/2), 6.4.1(5), 12.4(7), 13.1.1(6/3), 13.11(15/5), 13.11.3(4/3)
object-oriented programming (OOP)
See dispatching operations of tagged types 3.9.2(1/5)
See tagged types and type extensions 3.9(1)
object_declaration 3.3.1(2/3)
used 3.1(3/3), 4.5.9(3/5), P.1
object_renaming_declaration 8.5.1(2/5)
used 4.5.9(3/5), 8.5(2), P.1
Object_Size attribute 13.3(58.2/5)
obsolescent feature J(1/2)
occur immediately within 8.1(13)
occurrence
of an interrupt C.3(2)
occurrence (of an exception) 11(1.c/5)
octal
literal 2.4.2(1)
octal literal 2.4.2(1)
Old attribute 6.1.1(26/5)
one's complement
modular types 3.5.4(27)
one-dimensional array 3.6(12)
one-pass context_clauses 10.1.2(1.a)
only as a completion
entry_body 9.5.2(16)
OOP (object-oriented programming)
See dispatching operations of tagged types 3.9.2(1/5)
See tagged types and type extensions 3.9(1)
opaque type
See private types and private extensions 7.3(1)
Open
in Ada.Direct_IO A.8.4(7), A.8.4(18.11/5), A.8.4(18.4/5)
in Ada.Sequential_IO A.8.1(7), A.8.1(15.11/5), A.8.1(15.4/5)
in Ada.Streams.Stream_IO A.12.1(9), A.12.1(26.11/5), A.12.1(26.4/5)
in Ada.Text_IO A.10.1(10), A.10.1(85.11/5), A.10.1(85.4/5)
open alternative 9.7.1(14)
open entry 9.5.3(5)
of a protected object 9.5.3(7/3)
of a task 9.5.3(6/3)
operand
of a qualified_expression 4.7(3/5)
of a type_conversion 4.6(3)
operand interval G.2.1(6)
operand type
of a type_conversion 4.6(3)
operates on a type 3.2.3(1/2)
operation 3.2(10.a)
operational aspect 1.3.1(53/5), 13.1(8.1/3)
specifiable attributes 13.3(5/3)
operational item 13.1(1.1/1)

operative constituent 4.4(9.2/5)
operator 6.6(1)
& 4.5(1), 4.5.3(3)
* 4.5(1), 4.5.5(1)
** 4.5(1), 4.5.6(7)
+ 4.5(1), 4.5.3(1), 4.5.4(1)
- 4.5(1), 4.5.3(1), 4.5.4(1)
/ 4.5(1), 4.5.5(1)
/= 4.5(1), 4.5.2(1)
< 4.5(1), 4.5.2(1)
<= 4.5(1), 4.5.2(1)
= 4.5(1), 4.5.2(1)
> 4.5(1), 4.5.2(1)
>= 4.5(1), 4.5.2(1)
abs 4.5(1), 4.5.6(1)
ampersand 4.5(1), 4.5.3(3)
and 4.5(1), 4.5.1(2)
binary 4.5(9)
binary adding 4.5.3(1)
concatenation 4.5(1), 4.5.3(3)
divide 4.5(1), 4.5.5(1)
equal 4.5(1), 4.5.2(1)
equality 4.5.2(1)
exponentiation 4.5(1), 4.5.6(7)
greater than 4.5(1), 4.5.2(1)
greater than or equal 4.5(1), 4.5.2(1)
highest precedence 4.5.6(1)
less than 4.5(1), 4.5.2(1)
less than or equal 4.5(1), 4.5.2(1)
logical 4.5.1(2)
minus 4.5(1), 4.5.3(1), 4.5.4(1)
mod 4.5(1), 4.5.5(1)
multiply 4.5(1), 4.5.5(1)
multiplying 4.5.5(1)
not 4.5(1), 4.5.6(3)
not equal 4.5(1), 4.5.2(1)
or 4.5(1), 4.5.1(2)
ordering 4.5.2(1)
plus 4.5(1), 4.5.3(1), 4.5.4(1)
predefined 4.5(9)
relational 4.5.2(1)
rem 4.5(1), 4.5.5(1)
times 4.5(1), 4.5.5(1)
unary 4.5(9)
unary adding 4.5.4(1)
user-defined 6.6(1)
xor 4.5(1), 4.5.1(2)
operator precedence 4.5(1)
operator_symbol 6.1(9)
used 4.1(3), 4.1.3(3), 6.1(5), 6.1(11), P.1
optimization 11.5(29/5), 11.6(1/3)
Optimize pragma 2.8(23), L(23)
or else (short-circuit control form) 4.5(1), 4.5.1(1)
or operator 4.5(1), 4.5.1(2)
Ordered_FIFO_Queuing queuing policy D.4(7.1/5)
Ordered_Maps
child of Ada.Containers A.18.6(2/5)
Ordered_Sets
child of Ada.Containers A.18.9(2/5)
ordering operator 4.5.2(1)
ordinary file A.16(45/2)
ordinary fixed point type 3.5.9(1), 3.5.9(8/2)
ordinary_fixed_point_definition 3.5.9(3)
used 3.5.9(2), P.1
OSC
in Ada.Characters.Latin_1 A.3.3(19)
other_control 2.1(13.1/2)
other_format 2.1(10.3/2)
other_private_use 2.1(13.2/2)
other_surrogate 2.1(13.3/2)
others choice 4.3.3(6.b)
output A.6(1/5)
Output aspect 13.13.2(38/5)
Output attribute 13.13.2(19), 13.13.2(29)
Output clause 13.3(7/2), 13.13.2(38/5)
Output'Class aspect 13.13.2(38/5)
outside
a subtype 3.2(8/5)
overall interpretation
of a complete context 8.6(10)
Overflow_Check 11.5(16)
[*partial*] 3.5.4(20), 4.4(11), G.2.1(11), G.2.2(7), G.2.3(25), G.2.4(2), G.2.6(3/5)
Overlap
in Ada.Containers.Hashed_Sets A.18.8(38/2)
in Ada.Containers.Ordered_Sets A.18.9(39/2)
Overlaps_Storage attribute 13.3(73.6/3)
overload resolution 8.6(1/3)
overloadable 8.3(7)
overloaded 8.3(6)
enumeration literal 3.5.1(9)
overloading rules 1.1.2(26/3), 8.6(2)
overridable 8.3(9/1)
override 8.3(9/1), 12.3(17)
a primitive subprogram 3.2.3(7/2)
when implemented by 8.3(13.b/3), 9.1(9.2/3), 9.4(11.1/3)
overriding operation 1.3.2(2/5)
overriding_indicator 8.3.1(2/2)
used 3.9.3(1.1/3), 6.1(2/3), 6.3(2/3), 6.7(2/3), 6.8(2/4), 8.5.4(2/3), 9.5.2(2/3), 10.1.3(3/3), 12.3(2/3), P.1
Overwrite
in Ada.Strings.Bounded A.4.4(62), A.4.4(63)
in Ada.Strings.Fixed A.4.3(27), A.4.3(28)
in Ada.Strings.Unbounded A.4.5(57), A.4.5(58)

P

Pack aspect 13.2(5.1/3)
Pack pragma J.15.3(2/3), L(24.1/3)
package 1.3.3(24/5), 7(1)
package instance 12.3(13)
package-private extension 7.3(14.a)
package-private type 7.3(14.a)
package_body 7.2(2/3)
used 3.11(6), 10.1.1(7), P.1
package_body_stub 10.1.3(4/3)
used 10.1.3(2), P.1
package_declaration 7.1(2)
used 3.1(3/3), 10.1.1(5), P.1
package_renaming_declaration 8.5.3(2/3)
used 8.5(2), 10.1.1(6), P.1
package_specification 7.1(3/3)
used 7.1(2), 12.1(4), P.1
packed 13.2(5.1/3)
Packed_Decimal
in Interfaces.COBOL B.4(12/3)
Packed_Format
in Interfaces.COBOL B.4(26)
Packed_Signed
in Interfaces.COBOL B.4(27)
Packed_Unsigned
in Interfaces.COBOL B.4(27)
padding bits 13.1(7/5)
Page
in Ada.Text_IO A.10.1(39/5)
Page pragma 2.8(22), L(25)
page terminator A.10(7)
Page_Length
in Ada.Text_IO A.10.1(26/5)
Paragraph_Sign
in Ada.Characters.Latin_1 A.3.3(22)
parallel construct 1.3.3(26/5), 5.1(1/5)
parallel iterable container object 5.5.1(11/5)
parallel iterable container type 5.5.1(11/5)
parallel iterator 5.5.2(4/5)
parallel iterator object 5.5.1(6/5)
parallel iterator type 5.5.1(6/5)
parallel loop 5.5(8.1/5)
parallel processing
See parallel construct 5.1(1/5)
See task 9(1/5)
parallel_block_statement 5.6.1(2/5)
used 5.1(5/5), P.1
Parallel_Calls aspect 9.10.1(24/5)
Parallel_Iterator
in Ada.Iterator_Interfaces 5.5.1(4.1/5)
Parallel_Reduce(Reducer, Initial_Value) attribute 4.5.10(33/5)
Parallel_Reversible_Iterator
in Ada.Iterator_Interfaces 5.5.1(4.8/5)
parameter
explicitly aliased 6.1(23.1/3)
See formal parameter 6.1(17)
See generic formal parameter 12(1)
See also discriminant 3.7(1/2)
See also loop parameter 5.5(6/5)
parameter assigning back 6.4.1(17)
parameter copy back 6.4.1(17)
parameter mode 6.1(18/3)
parameter passing 6.4.1(1)
parameter_and_result_profile 6.1(13/2)
used 3.10(5), 3.10(6/2), 6.1(4.2/2), P.1
parameter_association 6.4(5)
used 5.5.3(6/5), 6.4(4), P.1
parameter_association_with_box 5.5.3(7/5)
used 5.5.3(6/5), P.1
parameter_profile 6.1(12)

used 3.10(5), 3.10(6/2), 6.1(4.1/2), 9.5.2(2/3), 9.5.2(3), 9.5.2(6), P.1
parameter_specification 6.1(15/5)
used 6.1(14), P.1
Parameterless_Handler
in Ada.Interrupts C.3.2(2/5)
Params_Stream_Type
in System.RPC E.5(6)
Parent
in Ada.Containers.Multiway_Trees A.18.10(59.1/5), A.18.10(59/5)
parent body
of a subunit 10.1.3(8/2)
parent declaration
of a library unit 10.1.1(10)
of a library_item 10.1.1(10)
parent of a derived type 1.3.1(54/5)
parent subtype 3.4(3/2)
parent type 3.4(3/2)
parent unit
of a library unit 10.1.1(10)
Parent_Tag
in Ada.Tags 3.9(7.2/2)
parent_unit_name 10.1.1(8)
used 6.1(5), 6.1(7), 7.1(3/3), 7.2(2/3), 10.1.3(7), P.1
parenthesized expression 4.4(9.1/5)
part
of a type 3.2(6/2)
of an object or value 3.2(6/2)
of the nominal subtype 3.9.1(4.2/5)
partial view 1.3.1(56/5)
of a type 7.3(4)
partition 1.3.3(27/5), 10.2(2/5)
partition building 10.2(2/5)
partition communication subsystem (PCS) E.5(1/2)
Partition_Check [*partial*] E.4(19)
Partition_Elaboration_Policy pragma H.6(3/2), L(25.1/2)
Partition_Id
in System.RPC E.5(4)
Partition_Id attribute E.1(9)
pass by copy 6.2(2)
pass by reference 6.2(2)
passive partition E.1(2)
Pattern_Error
in Ada.Strings A.4.1(5)
PC-map approach to finalization 7.6.1(24.s)
PCS (partition communication subsystem) E.5(1/2)
Peak_Use
in Ada.Containers.Bounded_Priority_Queues A.18.31(7/5)
in Ada.Containers.Bounded_Synchronized_Queues A.18.29(6/5)
in Ada.Containers.Synchronized_Queue_Interfaces A.18.27(7/5)
in Ada.Containers.Unbounded_Priority_Queues A.18.30(7/5)
in Ada.Containers.Unbounded_Synchronized_Queues A.18.28(6/5)
pending interrupt occurrence C.3(2)
per-object constraint 3.8(18/2)
per-object expression 3.8(18/2)
percent sign 2.1(15/5)
Percent_Sign
in Ada.Characters.Latin_1 A.3.3(8)
perfect result set G.2.3(5)
perform indefinite insertion A.18(6/5)
periodic task
example 9.6(39)
See delay_until_statement 9.6(39)
Pi
in Ada.Numerics A.5(3/5)
Pic_String
in Ada.Text_IO.Editing F.3.3(7)
Picture
in Ada.Text_IO.Editing F.3.3(4)
picture String
for edited output F.3.1(1/3)
Picture_Error
in Ada.Text_IO.Editing F.3.3(9)
Pilcrow_Sign
in Ada.Characters.Latin_1 A.3.3(22)
plain_char
in Interfaces.C B.3(11)
plane
character 2.1(1/5)
PLD
in Ada.Characters.Latin_1 A.3.3(17)
PLU
in Ada.Characters.Latin_1 A.3.3(17)
plus operator 4.5(1), 4.5.3(1), 4.5.4(1)
plus sign 2.1(15/5)
Plus_Minus_Sign
in Ada.Characters.Latin_1 A.3.3(22)
Plus_Sign
in Ada.Characters.Latin_1 A.3.3(8)
PM
in Ada.Characters.Latin_1 A.3.3(19)
point 2.1(15/5)
Pointer
in Interfaces.C.Pointers B.3.2(5)
See access value 3.10(1)
See type System.Address 13.7(34/2)
pointer type
See access type 3.10(1)
Pointer_Error
in Interfaces.C.Pointers B.3.2(8)
Pointers
child of Interfaces.C B.3.2(4/5)
polymorphism 3.9(1), 3.9.2(1/5)
pool
default 13.11.3(4.2/4)
subpool 13.11.4(18/3)
pool element 3.10(7/1), 13.11(11)
pool type 13.11(11)
pool-specific access type 3.10(7/1), 3.10(8)
Pool_of_Subpool
in System.Storage_Pools.Subpools 13.11.4(9/3)
Pos attribute 3.5.5(2)
position 13.5.1(4)
used 13.5.1(3), P.1
Position attribute 13.5.2(2/2)
position number 3.5(1)
of an enumeration value 3.5.1(7)
of an integer value 3.5.4(15)
positional association 6.4(7/5), 6.4.1(2/3), 12.3(6)
positional component association 4.3.1(6)
positional discriminant association 3.7.1(4)
positional parameter association 6.4.1(2/3)
positional_array_aggregate 4.3.3(3/5)
used 4.3.3(2/5), P.1
positional_container_aggregate 4.3.5(15/5)
used 4.3.5(13/5), P.1
Positive 3.5.4(12)
Positive *subtype of* Integer
in Standard A.1(13)
Positive_Count *subtype of* Count
in Ada.Direct_IO A.8.4(4)
in Ada.Streams.Stream_IO A.12.1(7)
in Ada.Text_IO A.10.1(5)
POSIX 1.2.1(10.a/5)
possible interpretation 8.6(14)
for direct_names 8.3(24)
for selector_names 8.3(24)
Post aspect 6.1.1(4/5)
Post'Class aspect 6.1.1(5/5)
post-compilation error 1.1.2(29)
post-compilation rules 1.1.2(29)
postcondition 1.3.2(4/5)
postcondition check 6.1.1(35/3)
postcondition expression
class-wide 6.1.1(5/5)
explicit 7.3.4(23/5)
specific 6.1.1(4/5)
potentially blocking operation 9.5(34/5)
delay_statement 9.6(34), D.9(5)
remote subprogram call E.4(17/5)
potentially conflict
actions 9.10(17/5)
potentially signal 9.10(10.1/5)
potentially static expression 6.8(3.2/5)
potentially use-visible 8.4(8/3) [*partial*] 12.6(9.2/3)
Pound_Sign
in Ada.Characters.Latin_1 A.3.3(21/3)
pragma 1.3.3(29/5), 2.8(1), 2.8(2), L(1)
pragma argument 2.8(9)
pragma name 2.8(9)
pragma, categorization
Pure J.15.15(9/5)
Remote_Call_Interface J.15.15(9/5)
Remote_Types J.15.15(9/5)
Shared_Passive J.15.15(9/5)
pragma, configuration 10.1.5(8)
Assertion_Policy 11.4.2(7/3)

Detect_Blocking H.5(4/2)
Discard_Names C.5(4)
Locking_Policy D.3(5)
Normalize_Scalars H.1(4)
Partition_Elaboration_Policy H.6(5/2)
Priority_Specific_Dispatching D.2.2(4/2)
Profile 13.12(14/3)
Queuing_Policy D.4(5)
Restrictions 13.12(8/3)
Reviewable H.3.1(4)
Suppress 11.5(5/2)
Task_Dispatching_Policy D.2.2(4/2)
Unsuppress 11.5(5/2)
pragma, identifier specific to 2.8(10/3)
pragma, interfacing
Convention J.15.5(1/3)
Export J.15.5(1/3)
Import J.15.5(1/3)
pragma, library unit J.15(7/5)
All_Calls_Remote J.15.15(7/5)
Elaborate_Body J.15.14(7/5)
Preelaborate J.15.14(7/5)
Pure J.15.14(7/5)
Remote_Call_Interface J.15.15(7/5)
Remote_Types J.15.15(7/5)
Shared_Passive J.15.15(7/5)
pragma, program unit J.15(2/5)
Inline J.15.1(1/3)
library unit pragmas J.15(7/5)
pragma, representation 13.1(1/1)
Asynchronous J.15.13(1/3)
Atomic J.15.8(9/3)
Atomic_Components J.15.8(9/3)
Convention J.15.5(1/3)
Discard_Names C.5(6)
Export J.15.5(1/3)
Import J.15.5(1/3)
Independent J.15.8(9/3)
Independent_Components J.15.8(9/3)
No_Return J.15.2(1/3)
Pack J.15.3(1/3)
Unchecked_Union J.15.6(1/3)
Volatile J.15.8(9/3)
Volatile_Components J.15.8(9/3)
pragma_argument_association 2.8(3/3)
used 2.8(2), 13.12(11/3), L(27.3/3), P.1
pragmas
Admission_Policy D.4.1(3/5), L(1.1/5)
All_Calls_Remote J.15.15(6/5), L(2.1/5)
Assert 11.4.2(3/2), L(2.2/2)
Assertion_Policy 11.4.2(6.1/3), 11.4.2(6/2), L(2.3/2), L(2.4/3)
Asynchronous J.15.13(2/3), L(3.1/3)
Atomic J.15.8(2/3), L(4.1/3)
Atomic_Components J.15.8(5/3), L(5.1/3)
Attach_Handler J.15.7(4/3), L(6.1/3)
Conflict_Check_Policy 9.10.1(3/5), L(6.2/5)
Convention J.15.5(4/3), L(8.1/3)
CPU J.15.9(2/3), L(8.2/3)
Default_Storage_Pool 13.11.3(3/3), L(8.3/3)
Detect_Blocking H.5(3/2), L(8.4/2)
Discard_Names C.5(3), L(9)
Dispatching_Domain J.15.10(2/3), L(9.1/3)
Elaborate 10.2.1(20), L(10)
Elaborate_All 10.2.1(21), L(11)
Elaborate_Body J.15.14(6/5), L(12.1/5)
Export J.15.5(3/3), L(13.1/3)
Generate_Deadlines D.2.6(9.4/5), L(13.2/5)
Import J.15.5(2/3), L(14.1/3)
Independent J.15.8(4/3), L(14.2/3)
Independent_Components J.15.8(7/3), L(14.3/3)
Inline J.15.1(2/3), L(15.1/3)
Inspection_Point H.3.2(3), L(16)
Interrupt_Handler J.15.7(2/3), L(17.1/3)
Interrupt_Priority J.15.11(4/3), L(18.1/3)
Linker_Options B.1(8), L(19)
List 2.8(21), L(20)
Locking_Policy D.3(3), L(21)
No_Return J.15.2(2/5), L(21.2/3)
Normalize_Scalars H.1(3), L(22)
Optimize 2.8(23), L(23)
Pack J.15.3(2/3), L(24.1/3)
Page 2.8(22), L(25)
Partition_Elaboration_Policy H.6(3/2), L(25.1/2)
Preelaborable_Initialization J.15.14(4/5), L(25.3/5)
Preelaborate J.15.14(3/5), L(26.1/5)
Priority J.15.11(2/3), L(27.1/3)
Priority_Specific_Dispatching D.2.2(2.2/2), L(27.2/2)
Profile 13.12(11/3), L(27.3/3)
Pure J.15.14(5/5), L(28.1/5)
Queuing_Policy D.4(3), L(29)
Relative_Deadline J.15.12(2/3), L(29.2/3)
Remote_Call_Interface J.15.15(5/5), L(30.1/5)
Remote_Types J.15.15(4/5), L(31.1/5)
Restrictions 13.12(3), L(32)
Reviewable H.3.1(3), L(33)
Shared_Passive J.15.15(3/5), L(34.1/5)
Storage_Size J.15.4(2/3), L(35.1/3)
Suppress 11.5(4/2), J.10(3/2), L(36)
Task_Dispatching_Policy D.2.2(2), L(37)
Unchecked_Union J.15.6(2/3), L(37.2/3)
Unsuppress 11.5(4.1/2), L(37.3/2)
Volatile J.15.8(3/3), L(38.1/3)
Volatile_Components J.15.8(6/3), L(39.1/3)
Pre aspect 6.1.1(2/5)
Pre'Class aspect 6.1.1(3/5)
precedence of operators 4.5(1)
precondition 1.3.2(5/5)
precondition check
class-wide 6.1.1(33/3)
specific 6.1.1(32/3)
precondition expression
class-wide 6.1.1(3/5)
specific 6.1.1(2/5)
Pred attribute 3.5(25)
predecessor element
of an ordered set A.18.9(81/3)
predecessor node
of an ordered map A.18.6(58/3)
predefined environment A(1)
predefined exception 11.1(4)
predefined library unit
See language-defined library units
predefined operation
of a type 3.2.3(1/2)
predefined operations
of a discrete type 3.5.5(10)
of a fixed point type 3.5.10(17)
of a floating point type 3.5.8(3)
of a record type 3.8(24)
of an access type 3.10.2(34/2)
of an array type 3.6.2(15)
predefined operator 4.5(9)
[*partial*] 3.2.1(9/5)
predefined type 3.2.1(10)
See language-defined types
predicate 4.5.8(3/3)
used 4.5.8(1/3), P.1
predicate aspect 3.2.4(1/5)
predicate check
allocator 3.2.4(31/5)
enabled 3.2.4(7/3)
in out parameters 3.2.4(31/5)
object_declaration 3.2.4(31/5)
subtype conversion 4.6(51/5)
predicate specification 3.2.4(1/5)
predicate-static 3.2.4(15/3)
Predicate_Failure aspect 3.2.4(14.2/4)
predicates satisfied 3.2.4(29.2/4)
predicates satisfied required membership 4.5.2(29/4)
Valid attribute 13.9.2(3/4), K.2(263/4)
preelaborable
of an elaborable construct 10.2.1(5)
preelaborable initialization 10.2.1(11.1/5)
Preelaborable_Initialization attribute 10.2.1(11.7/5)
Preelaborable_Initialization pragma J.15.14(4/5), L(25.3/5)
Preelaborate aspect 10.2.1(11/5)
Preelaborate pragma J.15.14(3/5), L(26.1/5)
preelaborated 10.2.1(11/5)
[*partial*] 10.2.1(11/5), E.2.1(9)
implementation requirements C.4(2)
preempt
a running task D.2.3(9/2)
preference
for root numeric operators and ranges 8.6(29)
for universal access equality operators 8.6(29.1/3)

preference control
  See requeue 9.5.4(1)
prefix 4.1(4)
  of a prefixed view 4.1.3(9.2/3)
  used 4.1.1(2), 4.1.2(2), 4.1.3(2), 4.1.4(2/5), 4.1.4(4), 4.1.6(10/3), 4.5.10(2/5), 5.5.3(4/5), 6.4(2), 6.4(3), P.1
prefixed view 4.1.3(9.2/3)
prefixed view profile 6.3.1(24.1/2)
Prepend
  in Ada.Containers.Doubly_Linked_-Lists A.18.3(22/5)
  in Ada.Containers.Vectors A.18.2(45/5)
Prepend_Child
  in Ada.Containers.Multiway_Trees A.18.10(51/5)
Prepend_Vector
  in Ada.Containers.Vectors A.18.2(44/5)
prescribed result
  for the evaluation of a complex arithmetic operation G.1.1(42)
  for the evaluation of a complex elementary function G.1.2(35)
  for the evaluation of an elementary function A.5.1(37)
Previous
  in Ada.Containers.Doubly_Linked_-Lists A.18.3(38.1/5), A.18.3(38/5), A.18.3(40.1/5), A.18.3(40/5)
  in Ada.Containers.Ordered_Maps A.18.6(36.1/5), A.18.6(36/5), A.18.6(37.1/5), A.18.6(37/5)
  in Ada.Containers.Ordered_Sets A.18.9(47.1/5), A.18.9(47/5), A.18.9(48.1/5), A.18.9(48/5)
  in Ada.Containers.Vectors A.18.2(65.1/5), A.18.2(65/5), A.18.2(66.1/5), A.18.2(66/5)
  in Ada.Iterator_Interfaces 5.5.1(4/3)
Previous_Sibling
  in Ada.Containers.Multiway_Trees A.18.10(66.1/5), A.18.10(66/5), A.18.10(67.1/5), A.18.10(67/5)
primary 4.4(7/5)
  used 4.4(6), P.1
primitive
  by-reference 13.1(10/5)
primitive function A.5.3(17)
primitive operation
  [*partial*] 3.2(1)
primitive operations
  of a type 3.2.3(1/2)
primitive operations of a type 1.3.1(58/5)
primitive operator
  of a type 3.2.3(8)
primitive subprograms
  of a type 3.2.3(2)
priority D.1(15/5)
  of a protected object D.3(6/2)
Priority aspect D.1(6.2/3)
Priority attribute D.5.2(3/2)
priority inheritance D.1(15/5)
priority inversion D.2.3(11/2)
priority of an entry call D.4(9)
Priority pragma J.15.11(2/3), L(27.1/3)
Priority *subtype of* Any_Priority
  in System 13.7(16)
Priority_Queuing queuing policy D.4(8)
Priority_Specific_Dispatching pragma D.2.2(2.2/2), L(27.2/2)
private declaration of a library unit 10.1.1(12)
private descendant
  of a library unit 10.1.1(12)
private extension 1.3.1(60/5), 3.2(4.1/2), 3.9(2.1/2), 3.9.1(1/2)
  [*partial*] 7.3(14), 12.5.1(5/3)
private library unit 10.1.1(12)
private operations 7.3.1(1)
private part 8.2(5)
  of a package 7.1(6/2), 12.3(12.b)
  of a protected unit 9.4(11/2)
  of a task unit 9.1(9)
private type 1.3.1(61/5), 3.2(4.1/2)
  [*partial*] 7.3(14)
private types and private extensions 7.3(1)
private with_clause 10.1.2(4.b/2)
private_extension_declaration 7.3(3/3)
  used 3.2.1(2), P.1
private_type_declaration 7.3(2/3)
  used 3.2.1(2), P.1
procedural_iterator 5.5.3(2/5)
  used 5.5(3/5), P.1
procedure 1.3.2(6/5), 6(1)
  null 6.7(3/5)
procedure instance 12.3(13)
procedure_call_statement 6.4(2)
  used 5.1(4/2), 9.7.2(3.1/2), P.1
procedure_or_entry_call 9.7.2(3.1/2)
  used 9.7.2(3/2), 9.7.4(4/2), P.1
procedure_specification 6.1(4.1/2)
  used 6.1(4/2), 6.7(2/3), P.1
processing node E(2)
profile 6.1(22)
  associated with a dereference 4.1(10)
  fully conformant 6.3.1(18/3)
  mode conformant 6.3.1(16/3)
  No_Implementation_Extensions 13.12.1(10/3)
  subtype conformant 6.3.1(17/3)
  type conformant 6.3.1(15/2)
Profile pragma 13.12(11/3), L(27.3/3)
profile resolution rule
  name with a given expected profile 8.6(26)
progenitor 1.3.1(63/5)
progenitor subtype 3.9.4(9/2)
progenitor type 3.9.4(9/2)
program 1.3.3(31/5), 10.2(1)
program execution 10.2(1)
program library
  See library 10(2)
  See library 10.1.4(9/5)
program unit 1.3.3(32/5), 10.1(1)
program unit pragma J.15(2/5)
  Inline J.15.1(1/3)
  library unit pragmas J.15(7/5)
program-counter-map approach to finalization 7.6.1(24.s)
Program_Error
  raised by a nonreturning procedure 6.5.1(9/2)
  raised by Address of an intrinsic subprogram 13.3(11.1/3)
  raised by barrier failure 9.5.3(7/3)
  raised by closed alternatives 9.7.1(21)
  raised by detection of a bounded error 1.1.5(8/5), 4.2.1(17/5), 4.5.10(34/5), 4.8(11.1/2), 5.1(18/5), 5.5.3(28/5), 5.5.3(29/5), 6.2(12/3), 8.5.4(8.1/1), 9.4(20.1/2), 9.5.1(17/5), 9.5.1(18/5), 9.8(20/5), 10.2(26), 11.4.2(23.1/5), 13.9.1(9), 13.11.2(13), 13.11.2(15.2/4), A.18.2(238/3), A.18.2(239.1/3), A.18.2(239.2/3), A.18.2(239/2), A.18.2(247/2), A.18.3(152.1/3), A.18.3(152.2/3), A.18.3(152/2), A.18.4(75.7/3), A.18.4(75.8/3), A.18.7(96.19/3), A.18.7(96.20/3), A.18.10(220/3), A.18.10(221/3), A.18.18(68/3), A.18.18(69/3), A.18.19(10/3), A.18.20(14/3), A.18.21(15/3), A.18.22(12/3), A.18.23(15/3), A.18.24(12/3), A.18.25(14/3), A.18.32(11/5), C.7.1(17/3), D.3(13.7/5), D.10(10.2/5), D.10.1(15/3)
  raised by deteection of a bounded error H.5(5/5)
  raised by failed finalization 7.6.1(15), 7.6.1(16/5), 7.6.1(17), 7.6.1(17.2/1), 7.6.1(18/2)
  raised by failure of runtime check 3.2.4(29.1/4), 3.10.2(29), 3.11(14), 4.6(57/5), 4.8(10.1/3), 4.8(10.2/2), 4.8(10.3/2), 4.8(10.4/3), 5.5(8.1/5), 5.6.1(4/5), 6.5(8/5), 6.5(21/3), 7.6.1(20.b), 9.5.4(7.2/5), 11.1(4), 11.5(8.a), 11.5(19/5), 12.5.1(23.3/2), 13.3(75.1/3), 13.11.4(27/3), 13.11.4(30/3), B.3.3(22/2), C.3.1(10/3), C.3.1(11/3), C.3.1(13.1/5), C.3.2(20), D.3(13), D.3(13.5/5), D.4(22/5), D.4(23/5), D.5.2(6/3), D.7(7.1/3), D.7(10.5/3), D.7(19.1/2), E.4(18/1), H.4(23.11/5), J.7.1(7)
  raised by finalization of a protected object 9.4(20)
  raised for missing return statement 6.4(11/2)
  in Standard A.1(46)
Program_Error_Check 11.5(21/5)
  [*partial*] 3.2.4(29.1/4), 5.5(8.1/5), 5.6.1(4/5), 12.5.1(23.3/2), 13.3(75.1/3), B.3.3(22/2), H.4(23.11/5)

prohibited
  tampering with a holder A.18.18(29/5)
  tampering with a list A.18.3(61/5)
  tampering with a map A.18.4(7/5)
  tampering with a set A.18.7(7/5)
  tampering with a tree A.18.10(80/5)
  tampering with a vector A.18.2(90/5)
propagate 11.4(1)
  an exception by a construct 11.4(6.a)
  an exception by an execution 11.4(6.a)
  an exception occurrence by an execution, to a dynamically enclosing execution 11.4(6)
proper_body 3.11(6)
  used 3.11(5), 10.1.3(7), P.1
property
  stable 7.3.4(1/5)
property function 7.3.4(2/5)
protected action 9.5.1(4/4)
  complete 9.5.1(6)
  start 9.5.1(5/4)
protected calling convention 6.3.1(12/4)
protected declaration 9.4(1)
protected entry 9.4(1)
protected function 9.5.1(1)
protected interface 3.9.4(5/2)
protected object 9(3), 9.4(1)
protected operation 9.4(1)
protected procedure 9.5.1(1)
protected subprogram 9.4(1), 9.5.1(1)
protected tagged type 3.9.4(6/2)
protected type 1.3.1(65/5)
protected unit 9.4(1)
protected_body 9.4(7/3)
  used 3.11(6), P.1
protected_body_stub 10.1.3(6/3)
  used 10.1.3(2), P.1
protected_definition 9.4(4)
  used 9.4(2/3), 9.4(3/3), P.1
protected_element_declaration 9.4(6)
  used 9.4(4), P.1
protected_operation_declaration 9.4(5/1)
  used 9.4(4), 9.4(6), P.1
protected_operation_item 9.4(8/4)
  used 9.4(7/3), P.1
protected_type_declaration 9.4(2/3)
  used 3.2.1(3/3), P.1
ptrdiff_t
  in Interfaces.C B.3(12)
PU1
  in Ada.Characters.Latin_1 A.3.3(18)
PU2
  in Ada.Characters.Latin_1 A.3.3(18)
public declaration of a library unit 10.1.1(12)
public descendant
  of a library unit 10.1.1(12)
public library unit 10.1.1(12)
punctuation_connector 2.1(10.2/2)
  used 2.3(3.1/3), P.1
pure 10.2.1(15.1/5)
Pure pragma J.15.14(5/5), L(28.1/5)
pure-barrier-eligible D.7(1.1/5)
Pure_Barriers restriction D.7(10.11/5)
Put
  in Ada.Strings.Text_Buffers A.4.12(7/5)
  in Ada.Text_IO A.10.1(42/5), A.10.1(48/5), A.10.1(55/5), A.10.1(60/5), A.10.1(66/5), A.10.1(67/5), A.10.1(71/5), A.10.1(72/5), A.10.1(76/5), A.10.1(77/5), A.10.1(82/5), A.10.1(83/5)
  in Ada.Text_IO.Bounded_IO A.10.11(4/2), A.10.11(5/2)
  in Ada.Text_IO.Complex_IO G.1.3(7), G.1.3(8/5)
  in Ada.Text_IO.Editing F.3.3(14/5), F.3.3(15/5), F.3.3(16)
  in Ada.Text_IO.Unbounded_IO A.10.12(4/2), A.10.12(5/2)
Put_Image
  in Ada.Numerics.Big_Numbers.Big_Integers A.5.6(17/5)
  in Ada.Numerics.Big_Numbers.Big_Reals A.5.7(20/5)
Put_Image aspect 4.10(4/5)
Put_Image attribute 4.10(3/5)
Put_Line
  in Ada.Text_IO A.10.1(50/5)
  in Ada.Text_IO.Bounded_IO A.10.11(6/2), A.10.11(7/2)
  in Ada.Text_IO.Unbounded_IO A.10.12(6/2), A.10.12(7/2)
Put_UTF_8
  in Ada.Strings.Text_Buffers A.4.12(10/5)

Q

qualified_expression 4.7(2)
  used 4.1(2/5), 4.8(2/3), 13.8(2), P.1
quantified expression 4.5.8(0.1/4)
quantified_expression 4.5.8(1/3)
  used 4.4(7/5), P.1
quantifier 4.5.8(2/3)
  used 4.5.8(1/3), P.1
Query_Element
  in Ada.Containers.Doubly_Linked_Lists A.18.3(16.1/5), A.18.3(16/5)
  in Ada.Containers.Hashed_Maps A.18.5(16.1/5), A.18.5(16/5)
  in Ada.Containers.Hashed_Sets A.18.8(17.1/5), A.18.8(17/5)
  in Ada.Containers.Indefinite_Holders A.18.18(14/5)
  in Ada.Containers.Multiway_Trees A.18.10(26.1/5), A.18.10(26/5)
  in Ada.Containers.Ordered_Maps A.18.6(15.1/5), A.18.6(15/5)
  in Ada.Containers.Ordered_Sets A.18.9(16.1/5), A.18.9(16/5)
  in Ada.Containers.Vectors A.18.2(31/5), A.18.2(32.1/5), A.18.2(32/5)
Question
  in Ada.Characters.Latin_1 A.3.3(10)
Queue
  in Ada.Containers.Bounded_Priority_Queues A.18.31(4/3)
  in Ada.Containers.Bounded_Synchronized_Queues A.18.29(4/3)
  in Ada.Containers.Synchronized_Queue_Interfaces A.18.27(4/3)
  in Ada.Containers.Unbounded_Priority_Queues A.18.30(4/3)
  in Ada.Containers.Unbounded_Synchronized_Queues A.18.28(4/3)
queuing policy D.4(1/5), D.4(6)
  FIFO_Queuing D.4(7/5)
  Ordered_FIFO_Queuing D.4(7.1/5)
  Priority_Queuing D.4(8)
Queuing_Policy pragma D.4(3), L(29)
Quotation
  in Ada.Characters.Latin_1 A.3.3(8)
quotation mark 2.1(15/5)
quoted string
  See string_literal 2.6(1)

R

raise
  an exception 11(1/3)
  an exception 11.3(4/4)
  an exception occurrence 11.4(3)
raise an exception 1.3.5(5/5)
Raise_Exception
  in Ada.Exceptions 11.4.1(4/3)
raise_expression 11.3(2.1/4)
  used 4.4(3/4), P.1
raise_statement 11.3(2/2)
  used 5.1(4/2), P.1
Random
  in Ada.Numerics.Discrete_Random A.5.2(20/5)
  in Ada.Numerics.Float_Random A.5.2(8/5)
random number A.5.2(1)
range 3.5(3), 3.5(4)
  of a scalar subtype 3.5(7)
  used 3.5(2), 3.6(6), 3.6.1(3), 3.8.1(5/3), 4.4(3.2/4), P.1
Range attribute 3.5(14), 3.6.2(7)
Range(N) attribute 3.6.2(8)
range_attribute_designator 4.1.4(5)
  used 4.1.4(4), P.1
range_attribute_reference 4.1.4(4)
  used 3.5(3), P.1
Range_Check 11.5(17/5)

[*partial*] 3.2.2(11), 3.5(24), 3.5(27), 3.5(39.12/3), 3.5(39.4/3), 3.5(39.5/3), 3.5(43/5), 3.5(55/5), 3.5.5(7), 3.5.9(19/4), 4.2(11/5), 4.3.3(28), 4.5.1(8), 4.5.6(6), 4.5.6(13), 4.6(28), 4.6(38), 4.6(46), 4.6(51/5), 4.7(4/5), 13.13.2(35/3), A.5.3(26), A.5.3(29), A.5.3(50), A.5.3(53), A.5.3(59), A.5.3(62), K.2(11), K.2(114), K.2(122), K.2(184), K.2(220), K.2(241), K.2(41), K.2(47)
range_constraint 3.5(2)
used 3.2.2(6), 3.5.9(5/4), J.3(2/4), P.1
Ravenscar D.13(1/5)
RCI
generic E.2.3(7/5)
library unit E.2.3(7/5)
package E.2.3(7/5)
Re
in Ada.Numerics.Generic_Complex_-Arrays G.3.2(7/2), G.3.2(27/2)
in Ada.Numerics.Generic_Complex_-Types G.1.1(6)
re-raise statement 11.3(3/4)
read
the value of an object 3.3(14)
in Ada.Direct_IO A.8.4(12/5)
in Ada.Sequential_IO A.8.1(12/5)
in Ada.Storage_IO A.9(6)
in Ada.Streams 13.13.1(5)
in Ada.Streams.Storage.Bounded 13.13.1(28/5)
in Ada.Streams.Storage.Unbounded 13.13.1(20/5)
in Ada.Streams.Stream_IO A.12.1(15/5), A.12.1(16/5)
in System.RPC E.5(7)
Read aspect 13.13.2(38/5)
Read attribute 13.13.2(6), 13.13.2(14)
Read clause 13.3(7/2), 13.13.2(38/5)
Read'Class aspect 13.13.2(38/5)
ready
a task state 9(10/5)
ready queue D.2.1(5/2)
ready task D.2.1(5/2)
Real
in Interfaces.Fortran B.5(6)
real literal 2.4(1)
real literals 3.5.6(4)
real time D.8(18)
real type 1.3.1(66/5), 3.2(3), 3.5.6(1)
real-time systems C(1), D(1/5)
Real_Arrays
child of Ada.Numerics G.3.1(31/2)
Real_Literal aspect 4.2.1(4/5)
Real_Matrix
in Ada.Numerics.Generic_Real_Arrays G.3.1(4/2)
real_range_specification 3.5.7(3)
used 3.5.7(2), 3.5.9(3), 3.5.9(4), P.1
Real_Time
child of Ada D.8(3/5)
real_type_definition 3.5.6(2)
used 3.2.1(4/2), P.1
Real_Vector
in Ada.Numerics.Generic_Real_Arrays G.3.1(4/2)
receiving stub E.4(10)
reclamation of storage 13.11.2(1)
recommended level of support 13.1(20/3)
Address attribute 13.3(15)
Alignment attribute for objects 13.3(33)
Alignment attribute for subtypes 13.3(29)
aspect Pack 13.2(7/3)
bit ordering 13.5.3(7)
Component_Size attribute 13.3(71)
enumeration_representation_clause 13.4(9)
record_representation_clause 13.5.1(17)
required in Systems Programming Annex C.2(2)
Size attribute 13.3(42/5), 13.3(54)
Stream_Size attribute 13.13.2(1.7/2)
unchecked conversion 13.9(16)
with respect to nonstatic expressions 13.1(21/3)
record 3.8(1)
explicitly limited 3.8(13.1/3)
record extension 1.3.1(68/5), 3.4(5/2), 3.9.1(1/2)
Record layout aspect 13.5(1)
record type 1.3.1(69/5), 3.8(1)
record_aggregate 4.3.1(2)
used 4.3(2/5), 13.8(14.b), P.1
record_component_association 4.3.1(4/2)
used 4.3.1(3), P.1
record_component_association_list 4.3.1(3)
used 4.3.1(2), 4.3.2(2), 4.3.4(3/5), P.1
record_definition 3.8(3/5)
used 3.8(2), 3.9.1(2), P.1
record_delta_aggregate 4.3.4(3/5)
used 4.3.4(2/5), P.1
record_extension_part 3.9.1(2)
used 3.4(2/2), P.1
record_representation_clause 13.5.1(2/5)
used 13.1(2/1), P.1
record_type_definition 3.8(2)
used 3.2.1(4/2), P.1
Reduce(Reducer, Initial_Value) attribute 4.5.10(23/5), 4.5.10(32/5)
reducer subprogram 4.5.10(11/5)
reduction expression 1.3.3(34/5), 4.5.10(1/5)
reduction_attribute_designator 4.5.10(4/5)
used 4.5.10(2/5), P.1
reduction_attribute_reference 4.5.10(2/5)
used 4.1.4(2/5), P.1
reduction_specification 4.5.10(5/5)
used 4.5.10(4/5), P.1
reference
dangling 13.11.2(15.1/4)
in Ada.Containers.Doubly_Linked_-Lists A.18.3(17.4/5)
in Ada.Containers.Hashed_Maps A.18.5(17.4/5), A.18.5(17.6/5)
in Ada.Containers.Indefinite_Holders A.18.18(19/5)
in Ada.Containers.Multiway_Trees A.18.10(31/5)
in Ada.Containers.Ordered_Maps A.18.6(16.4/5), A.18.6(16.6/5)
in Ada.Containers.Vectors A.18.2(34.4/5), A.18.2(34.6/5)
in Ada.Interrupts C.3.2(10)
in Ada.Task_Attributes C.7.2(5)
reference discriminant 4.1.5(3/3)
reference object 4.1.5(3/3)
reference parameter passing 6.2(2)
reference type 1.3.1(70/5), 4.1.5(3/3)
Reference_Preserving_Key
in Ada.Containers.Hashed_Sets A.18.8(58.2/5), A.18.8(58.4/5)
in Ada.Containers.Ordered_Sets A.18.9(73.2/5), A.18.9(73.4/5)
Reference_Type
in Ada.Containers.Doubly_Linked_-Lists A.18.3(17.2/5), A.18.3(51.11/5)
in Ada.Containers.Hashed_Maps A.18.5(17.2/5), A.18.5(37.12/5)
in Ada.Containers.Hashed_Sets A.18.8(58.1/5)
in Ada.Containers.Indefinite_Holders A.18.18(17/5)
in Ada.Containers.Multiway_Trees A.18.10(29/5), A.18.10(70.11/5)
in Ada.Containers.Ordered_Maps A.18.6(16.2/5), A.18.6(51.13/5)
in Ada.Containers.Ordered_Sets A.18.9(73.1/5)
in Ada.Containers.Vectors A.18.2(34.2/5), A.18.2(79.11/5)
references 1.2(1/3)
Registered_Trade_Mark_Sign
in Ada.Characters.Latin_1 A.3.3(21/3)
Reinitialize
in Ada.Task_Attributes C.7.2(6)
relation 4.4(3/4)
used 4.4(2), P.1
relational operator 4.5.2(1)
relational_operator 4.5(3)
used 4.4(2.2/3), 4.4(3/4), P.1
relative deadline
protected object D.3(13.2/5)
Relative_Deadline aspect D.2.6(9.2/3)
Relative_Deadline attribute D.5.2(3.1/5)
Relative_Deadline pragma J.15.12(2/3), L(29.2/3)
Relative_Deadline *subtype of* Time_Span
in Ada.Dispatching.EDF D.2.6(9/5)
Relative_Name
in Ada.Directories.Hierarchical_File_Names A.16.1(13/3)
relaxed mode G.2(1)
release
execution resource associated with protected object 9.5.1(6)

rem operator 4.5(1), 4.5.5(1)
Remainder attribute A.5.3(45)
remote access E.1(5)
remote access type E.2.2(9/3)
remote access-to-class-wide type E.2.2(9/3)
remote access-to-subprogram type E.2.2(9/3)
remote call interface E.2(4/3), E.2.3(7/5)
remote procedure call
  asynchronous E.4.1(9/3)
remote subprogram E.2.3(7/5)
remote subprogram binding E.4(1/5)
remote subprogram call E.4(1/5)
remote types library unit E.2(4/3), E.2.2(4/5)
Remote_Call_Interface aspect E.2.3(7/5)
Remote_Call_Interface pragma J.15.15(5/5), L(30.1/5)
Remote_Types aspect E.2.2(4/5)
Remote_Types pragma J.15.15(4/5), L(31.1/5)
Remove_Task
  in Ada.Execution_Time.Group_Budgets D.14.2(8/2)
Rename
  in Ada.Directories A.16(12/2)
renamed entity 8.5(3)
renamed view 8.5(3)
renaming 1.3.3(35/5)
renaming-as-body 8.5.4(1/3)
renaming-as-declaration 8.5.4(1/3)
renaming_declaration 8.5(2)
  used 3.1(3/3), P.1
rendezvous 9.5.2(23.1/5)
repeatedly evaluated
  subexpression 6.1.1(22.12/5)
Replace
  in Ada.Containers.Hashed_Maps A.18.5(23/5)
  in Ada.Containers.Hashed_Sets A.18.8(22/5), A.18.8(53/5)
  in Ada.Containers.Ordered_Maps A.18.6(22/5)
  in Ada.Containers.Ordered_Sets A.18.9(21/5), A.18.9(66/5)
Replace_Element
  in Ada.Containers.Doubly_Linked_-Lists A.18.3(15/5)
  in Ada.Containers.Hashed_Maps A.18.5(15/5)
  in Ada.Containers.Hashed_Sets A.18.8(16/5)
  in Ada.Containers.Indefinite_Holders A.18.18(13/5)
  in Ada.Containers.Multiway_Trees A.18.10(25/5)
  in Ada.Containers.Ordered_Maps A.18.6(14/5)
  in Ada.Containers.Ordered_Sets A.18.9(15/5)
  in Ada.Containers.Vectors A.18.2(29/5), A.18.2(30/5)
  in Ada.Strings.Bounded A.4.4(27)
  in Ada.Strings.Unbounded A.4.5(21)
Replace_Slice
  in Ada.Strings.Bounded A.4.4(58), A.4.4(59)
  in Ada.Strings.Fixed A.4.3(23), A.4.3(24)
  in Ada.Strings.Unbounded A.4.5(53), A.4.5(54)
Replenish
  in Ada.Execution_Time.Group_Budgets D.14.2(9/2)
Replicate
  in Ada.Strings.Bounded A.4.4(78), A.4.4(79), A.4.4(80)
representation
  change of 13.6(1/5)
representation aspect 1.3.1(71/5), 13.1(8/5)
  coding 13.4(7)
  convention, calling convention B.1(1/3)
  export B.1(1/3)
  external_name B.1(1/3)
  import B.1(1/3)
  layout 13.5(1)
  link_name B.1(1/3)
  record layout 13.5(1)
  specifiable attributes 13.3(5/3)
  storage place 13.5(1)
representation attribute 13.3(1/1)
  Bit_Order 13.5.3(3/5)
representation item 13.1(1/1)
representation of an object 13.1(7/5)
representation pragma 13.1(1/1)
  Asynchronous J.15.13(1/3)
  Atomic J.15.8(9/3)
  Atomic_Components J.15.8(9/3)
  Convention J.15.5(1/3)
  Discard_Names C.5(6)
  Export J.15.5(1/3)
  Import J.15.5(1/3)
  Independent J.15.8(9/3)
  Independent_Components J.15.8(9/3)
  No_Return J.15.2(1/3)
  Pack J.15.3(1/3)
  Unchecked_Union J.15.6(1/3)
  Volatile J.15.8(9/3)
  Volatile_Components J.15.8(9/3)
representation value 13.4(10.4/5), K.2(59.4/5)
representation-oriented attributes
  of a fixed point subtype A.5.4(1)
  of a floating point subtype A.5.3(1)
representation_clause
  See aspect_clause 13.1(4/1)
represented in canonical form A.5.3(10)
requested decimal precision
  of a floating point type 3.5.7(4)
requeue 9.5.4(1)
requeue target 9.5.4(3/3)
requeue-with-abort 9.5.4(13)
requeue_statement 9.5.4(2/3)
  used 5.1(4/2), P.1
require overriding 3.9.3(6/5)
  [*partial*] 6.1.1(16/3)
requires a completion 3.11.1(1/3), 3.11.1(6/3)
  declaration for which aspect Elaborate_Body is True 10.2.1(25/5)
  declaration of a partial view 7.3(4)
  deferred constant declaration 7.4(2/3)
  generic_package_declaration 7.1(5/2)
  generic_subprogram_declaration 6.1(20/3)
  incomplete_type_declaration 3.10.1(3/3)
  library_unit_declaration 10.2(18.c/5)
  package_declaration 7.1(5/2)
  protected entry_declaration 9.5.2(16)
  protected_declaration 9.4(11.2/2)
  subprogram_declaration 6.1(20/3)
  task_declaration 9.1(9.3/2)
requires late initialization 3.3.1(8.1/5)
Reraise_Occurrence
  in Ada.Exceptions 11.4.1(4/3)
Reserve_Capacity
  in Ada.Containers.Hashed_Maps A.18.5(9/5)
  in Ada.Containers.Hashed_Sets A.18.8(11/5)
  in Ada.Containers.Vectors A.18.2(20/5)
reserved interrupt C.3(2)
reserved word 2.9(2/3)
Reserved_128
  in Ada.Characters.Latin_1 A.3.3(17)
Reserved_129
  in Ada.Characters.Latin_1 A.3.3(17)
Reserved_132
  in Ada.Characters.Latin_1 A.3.3(17)
Reserved_153
  in Ada.Characters.Latin_1 A.3.3(19)
Reserved_Check
  [*partial*] C.3.1(10/3)
Reset
  in Ada.Direct_IO A.8.4(8)
  in Ada.Numerics.Discrete_Random A.5.2(21/5), A.5.2(24/5)
  in Ada.Numerics.Float_Random A.5.2(9/5), A.5.2(12/5)
  in Ada.Sequential_IO A.8.1(8)
  in Ada.Streams.Stream_IO A.12.1(10)
  in Ada.Text_IO A.10.1(11)
resolution rules 1.1.2(26/3)
resolve
  overload resolution 8.6(14)
restriction 13.12(4/2)
  used 13.12(3), L(32)
restriction_parameter_argument 13.12(4.1/2)
  used 13.12(4/2), P.1
restrictions
  Immediate_Reclamation H.4(10)
  Max_Asynchronous_Select_Nesting D.7(18/1)
  Max_Entry_Queue_Length D.7(19.1/2)
  Max_Image_Length H.4(23.11/5)

Max_Protected_Entries D.7(14)
Max_Select_Alternatives D.7(12)
Max_Storage_At_Blocking D.7(17/1)
Max_Task_Entries D.7(13)
Max_Tasks D.7(19/1)
No_Abort_Statements D.7(5/3)
No_Access_Parameter_Allocators H.4(8.3/3)
No_Access_Subprograms H.4(17)
No_Allocators H.4(7)
No_Anonymous_Allocators H.4(8.1/3)
No_Asynchronous_Control J.13(3/2)
No_Coextensions H.4(8.2/3)
No_Delay H.4(21)
No_Dependence 13.12.1(6/2)
No_Dispatch H.4(19)
No_Dynamic_Attachment D.7(10/3)
No_Dynamic_CPU_Assignment D.7(10.1/4)
No_Dynamic_Priorities D.7(9/2)
No_Exceptions H.4(12/5)
No_Fixed_Point H.4(15)
No_Floating_Point H.4(14)
No_Hidden_Indirect_Globals H.4(23.2/5)
No_Implementation_Aspect_Specifications 13.12.1(1.1/3)
No_Implementation_Attributes 13.12.1(2/2)
No_Implementation_Identifiers 13.12.1(2.1/3)
No_Implementation_Pragmas 13.12.1(3/2)
No_Implementation_Units 13.12.1(3.1/3)
No_Implicit_Heap_Allocations D.7(8)
No_IO H.4(20/5)
No_Local_Allocators H.4(8/1)
No_Local_Protected_Objects D.7(10.2/3)
No_Local_Timing_Events D.7(10.3/3)
No_Nested_Finalization D.7(4/3)
No_Obsolescent_Features 13.12.1(4/3)
No_Protected_Type_Allocators D.7(10.4/2)
No_Protected_Types H.4(5)
No_Recursion H.4(22)
No_Reentrancy H.4(23)
No_Relative_Delay D.7(10.6/3)
No_Requeue_Statements D.7(10.7/3)
No_Select_Statements D.7(10.8/3)
No_Specific_Termination_Handlers D.7(10.9/3)
No_Specification_of_Aspect 13.12.1(6.1/3)
No_Standard_Allocators_After_Elaboration D.7(19.2/3)
No_Task_Allocators D.7(7)
No_Task_Hierarchy D.7(3/3)
No_Task_Termination D.7(15.1/2)
No_Tasks_Unassigned_To_CPU D.7(10.10/4)
No_Terminate_Alternatives D.7(6)
No_Unchecked_Access H.4(18)
No_Unchecked_Conversion J.13(4/2)
No_Unchecked_Deallocation J.13(5/2)
No_Unrecognized_Aspects 13.12.1(6.4/5)
No_Unrecognized_Pragmas 13.12.1(6.5/5)
No_Unspecified_Globals H.4(23.1/5)
No_Use_Of_Attribute 13.12.1(6.2/3)
No_Use_Of_Pragma 13.12.1(6.3/3)
Pure_Barriers D.7(10.11/5)
Simple_Barriers D.7(10.12/5)
Restrictions pragma 13.12(3), L(32)
Result attribute 6.1.1(29/5)
result interval
  for a component of the result of evaluating a complex function G.2.6(3/5)
  for the evaluation of a predefined arithmetic operation G.2.1(8)
  for the evaluation of an elementary function G.2.4(2)
result range
  of a Random function A.5.2(40.2/5)
result subtype
  of a function 6.5(3/5)
return expression
  of expression function 6.8(3.1/5)
return object
  extended_return_statement 6.5(5.11/5)
  simple_return_statement 6.5(6/2)
return statement 6.5(1/2)
return_subtype_indication 6.5(2.3/2)
  used 6.5(2.1/5), P.1
returns a readable reference H.4(23.7/5)
returns a writable reference H.4(23.7/5)
reverse iterator 5.5.2(4/5)
Reverse_Elements
  in Ada.Containers.Doubly_Linked_Lists A.18.3(27/5)
  in Ada.Containers.Vectors A.18.2(54/5)
Reverse_Find
  in Ada.Containers.Doubly_Linked_Lists A.18.3(42/5)
  in Ada.Containers.Vectors A.18.2(70/5)
Reverse_Find_Index
  in Ada.Containers.Vectors A.18.2(69/2)
Reverse_Iterate
  in Ada.Containers.Doubly_Linked_Lists A.18.3(46/5)
  in Ada.Containers.Ordered_Maps A.18.6(51/5)
  in Ada.Containers.Ordered_Sets A.18.9(61/5)
  in Ada.Containers.Vectors A.18.2(74/5)
Reverse_Iterate_Children
  in Ada.Containers.Multiway_Trees A.18.10(69.1/5), A.18.10(69/5)
Reverse_Solidus
  in Ada.Characters.Latin_1 A.3.3(12)
reversible iterable container object 5.5.1(11/5)
reversible iterable container type 5.5.1(11/5)
reversible iterator object 5.5.1(6/5)
reversible iterator type 5.5.1(6/5)
Reversible_Iterator
  in Ada.Iterator_Interfaces 5.5.1(4/3)
Reviewable pragma H.3.1(3), L(33)
RI
  in Ada.Characters.Latin_1 A.3.3(17)
right parenthesis 2.1(15/5)
right square bracket 2.1(15/5)
Right_Angle_Quotation
  in Ada.Characters.Latin_1 A.3.3(22)
Right_Curly_Bracket
  in Ada.Characters.Latin_1 A.3.3(14)
Right_Parenthesis
  in Ada.Characters.Latin_1 A.3.3(8)
Right_Square_Bracket
  in Ada.Characters.Latin_1 A.3.3(12)
Ring_Above
  in Ada.Characters.Latin_1 A.3.3(22)
ripple effect 10.1.2(1.b/2)
root
  of a tree A.18.10(2/4)
  in Ada.Containers.Multiway_Trees A.18.10(22/5)
root directory A.16(47.1/5)
root library unit 10.1.1(10)
root node
  of a tree A.18.10(2/4)
root type
  of a class 3.4.1(2/2)
Root_Buffer_Type
  in Ada.Strings.Text_Buffers A.4.12(6/5)
root_integer 3.5.4(14)
  [*partial*] 3.4.1(8)
root_real 3.5.6(3)
  [*partial*] 3.4.1(8)
Root_Storage_Pool
  in System.Storage_Pools 13.11(6/5)
Root_Storage_Pool_With_Subpools
  in System.Storage_Pools.Subpools 13.11.4(4/5)
Root_Stream_Type
  in Ada.Streams 13.13.1(3/2)
Root_Subpool
  in System.Storage_Pools.Subpools 13.11.4(5/5)
rooted at a type 3.4.1(2/2)
roots the subtree A.18.10(3/4)
Rosen trick 13.9.1(14.e/3)
rotate B.2(9)
Rotate_Left B.2(6)
Rotate_Right B.2(6)
Round attribute 3.5.10(12)
Round_Robin
  child of Ada.Dispatching D.2.5(4/5)
Round_Robin_Within_Priorities task dispatching policy D.2.5(2/2)
Rounding attribute A.5.3(36)

RPC
  child of System E.5(3/5)
RPC-receiver E.5(21)
RPC_Receiver
  in System.RPC E.5(11)
RS
  in Ada.Characters.Latin_1 A.3.3(6)
run-time error 1.1.2(30), 1.1.5(6), 11.5(2/3), 11.6(1/3)
run-time polymorphism 3.9.2(1/5)
run-time semantics 1.1.2(30)
run-time type
  See tag 3.9(3)
running a program
  See program execution 10.2(1)
running task D.2.1(6/2)
runtime check
  See language-defined check 11.5(2/3)
runtime name text
  entity with C.5(1.1/4)

S

safe range
  of a floating point type 3.5.7(9)
  of a floating point type 3.5.7(10/5)
safe separate compilation 10(3.b)
Safe_First attribute A.5.3(71), G.2.2(5)
Safe_Last attribute A.5.3(72), G.2.2(6)
safety-critical systems H(1/2)
same value
  for a limited type 6.2(10.f)
satisfied
  filter 5.5(6.2/5)
satisfies
  a discriminant constraint 3.7.1(11)
  a range constraint 3.5(4)
  an index constraint 3.6.1(7)
  for an access value 3.10(15/2)
satisfies the predicates
  of a subtype 3.2.4(29.2/4)
Saturday
  in Ada.Calendar.Formatting 9.6.1(17/2)
Save
  in Ada.Numerics.Discrete_Random A.5.2(24/5)
  in Ada.Numerics.Float_Random A.5.2(12/5)
Save_Occurrence
  in Ada.Exceptions 11.4.1(6/2)
scalar type 1.3.1(72/5), 3.2(3), 3.5(1)
scalar_constraint 3.2.2(6)
  used 3.2.2(5), P.1
scale
  of a decimal fixed point subtype 3.5.10(11), K.2(216)
Scale attribute 3.5.10(11)
Scaling attribute A.5.3(27)
SCHAR_MAX
  in Interfaces.C B.3(6)
SCHAR_MIN
  in Interfaces.C B.3(6)
SCI
  in Ada.Characters.Latin_1 A.3.3(19)
scope
  informal definition 3.1(8)
  of (a view of) an entity 8.2(11)
  of a declaration 8.2(10)
  of a use_clause 8.4(6)
  of a with_clause 10.1.2(5)
  of an aspect_specification 8.2(10.1/3)
  of an attribute_definition_clause 8.2(10.1/3)
Search_Type
  in Ada.Directories A.16(31/2)
Second
  in Ada.Calendar.Formatting 9.6.1(26/2)
Second_Duration *subtype of* Day_Duration
  in Ada.Calendar.Formatting 9.6.1(20/2)
Second_Number *subtype of* Natural
  in Ada.Calendar.Formatting 9.6.1(20/2)
Seconds
  in Ada.Calendar 9.6(13)
  in Ada.Real_Time D.8(14/2)
Seconds_Count
  in Ada.Real_Time D.8(15)
Seconds_Of
  in Ada.Calendar.Formatting 9.6.1(28/2)
Section_Sign
  in Ada.Characters.Latin_1 A.3.3(21/3)
secure systems H(1/2)
select an entry call
  from an entry queue 9.5.3(13), 9.5.3(16)
  immediately 9.5.3(8)
select_alternative 9.7.1(4)
  used 9.7.1(2), P.1
select_statement 9.7(2)
  used 5.1(5/5), P.1
selected_component 4.1.3(2)
  used 4.1(2/5), P.1
selection
  of an entry caller 9.5.2(24/5)
selective_accept 9.7.1(2)
  used 9.7(2), P.1
selector_name 4.1.3(3)
  used 3.7.1(3/5), 4.1.3(2), 4.3.1(5/5), 5.5.3(7/5), 6.4(5), 12.3(4), 12.7(3.1/2), P.1
semantic dependence
  of one compilation unit upon another 10.1.1(26/2)
semicolon 2.1(15/5)
  in Ada.Characters.Latin_1 A.3.3(10)
separate compilation 10.1(1)
  safe 10(3.b)
Separate_Interrupt_Clocks_Supported
  in Ada.Execution_Time D.14(9.2/3)
separator 2.2(3/2)
separator_line 2.1(12/2)
separator_paragraph 2.1(12.1/2)
separator_space 2.1(11/2)
sequence of characters
  of a string_literal 2.6(5)
sequence_of_statements 5.1(2/3)
  used 5.3(2), 5.4(3), 5.5(2), 5.6.1(2/5), 9.7.1(2), 9.7.1(5), 9.7.1(6), 9.7.2(3/2), 9.7.3(2), 9.7.4(3), 9.7.4(5), 11.2(2), 11.2(3/5), P.1
sequential
  actions 9.10(11/5), C.6(17)
sequential access A.8(2)
sequential file A.8(1/2)
sequential iterator 5.5.2(4/5)
sequential loop 5.5(9/5)
Sequential_IO
  child of Ada A.8.1(2/5)
service
  an entry queue 9.5.3(13)
set
  execution timer object D.14.1(12/2)
  group budget object D.14.2(15/2)
  termination handler C.7.3(9/2)
  timing event object D.15(9/2)
  in Ada.Containers.Hashed_Sets A.18.8(3/5), A.18.8(59.2/5)
  in Ada.Containers.Ordered_Sets A.18.9(4/5), A.18.9(74.2/5)
  in Ada.Environment_Variables A.17(6/2)
set container A.18.7(1/2)
Set_Bounded_String
  in Ada.Strings.Bounded A.4.4(12.1/2)
Set_Col
  in Ada.Text_IO A.10.1(35/5)
Set_CPU
  in System.Multiprocessors.Dispatching_Domains D.16.1(12/3)
Set_Deadline
  in Ada.Dispatching.EDF D.2.6(9/5)
Set_Dependents_Fallback_Handler
  in Ada.Task_Termination C.7.3(5/2)
Set_Directory
  in Ada.Directories A.16(6/2)
Set_Error
  in Ada.Text_IO A.10.1(15)
Set_Exit_Status
  in Ada.Command_Line A.15(9)
Set_False
  in Ada.Synchronous_Task_Control D.10(4/5)
Set_Handler
  in Ada.Execution_Time.Group_Budgets D.14.2(10/2)
  in Ada.Execution_Time.Timers D.14.1(7/2)
  in Ada.Real_Time.Timing_Events D.15(5/2)
Set_Im
  in Ada.Numerics.Generic_Complex_-Arrays G.3.2(8/2), G.3.2(28/2)
  in Ada.Numerics.Generic_Complex_-Types G.1.1(7)
Set_Index
  in Ada.Direct_IO A.8.4(14/5)
  in Ada.Streams.Stream_IO A.12.1(22/5)

Set_Input
  in Ada.Text_IO A.10.1(15)
Set_Iterator_Interfaces
  in Ada.Containers.Hashed_Sets A.18.8(6.3/3), A.18.8(59.7/5)
  in Ada.Containers.Ordered_Sets A.18.9(7.3/3), A.18.9(74.7/5)
Set_Length
  in Ada.Containers.Vectors A.18.2(22/5)
Set_Line
  in Ada.Text_IO A.10.1(36/5)
Set_Line_Length
  in Ada.Text_IO A.10.1(23/5)
Set_Mode
  in Ada.Streams.Stream_IO A.12.1(24)
Set_Output
  in Ada.Text_IO A.10.1(15)
Set_Page_Length
  in Ada.Text_IO A.10.1(24/5)
Set_Pool_of_Subpool
  in System.Storage_Pools.Subpools 13.11.4(10/5)
Set_Priority
  in Ada.Dynamic_Priorities D.5.1(4)
Set_Quantum
  in Ada.Dispatching.Round_Robin D.2.5(4/5)
Set_Re
  in Ada.Numerics.Generic_Complex_-Arrays G.3.2(8/2), G.3.2(28/2)
  in Ada.Numerics.Generic_Complex_-Types G.1.1(7)
Set_Relative_Deadline
  in Ada.Dispatching.EDF D.2.6(9/5)
Set_Specific_Handler
  in Ada.Task_Termination C.7.3(6/2)
Set_True
  in Ada.Synchronous_Task_Control D.10(4/5)
Set_Unbounded_String
  in Ada.Strings.Unbounded A.4.5(11.1/2)
Set_Value
  in Ada.Task_Attributes C.7.2(6)
shared passive library unit E.2(4/3), E.2.1(4/5)
shared variable
  protection of 9.10(1/5)
Shared_Passive aspect E.2.1(4/5)
Shared_Passive pragma J.15.15(3/5), L(34.1/5)
shift B.2(9)
Shift_Left B.2(6)
Shift_Right B.2(6)
Shift_Right_Arithmetic B.2(6)
short
  in Interfaces.C B.3(7)
short-circuit control form 4.5.1(1)
Short_Float 3.5.7(16/5)
Short_Integer 3.5.4(25)
SI
  in Ada.Characters.Latin_1 A.3.3(5)
signal
  as defined between actions 9.10(2/5)
  See interrupt C.3(1/3)
signal (an exception)
  See raise 11(1/3)
signal handling
  example 9.7.4(10)
signed integer type 3.5.4(1)
signed_char
  in Interfaces.C B.3(8)
Signed_Conversions
  in Ada.Numerics.Big_Numbers.Big_Integers A.5.6(12/5)
signed_integer_type_definition 3.5.4(3)
  used 3.5.4(2), P.1
Signed_Zeros attribute A.5.3(13)
simple entry call 9.5.3(1)
simple name
  of a file A.16(47/2)
Simple_Barriers restriction D.7(10.12/5)
simple_expression 4.4(4)
  used 3.5(3), 3.5.4(3), 3.5.7(3), 3.5.9(5/4), 4.4(2.2/3), 4.4(3.2/4), 4.4(3/4), 5.5(3.1/5), 11.3(2.1/4), 13.5.1(5), 13.5.1(6), J.3(2/4), P.1
Simple_Name
  in Ada.Directories A.16(16/5), A.16(38/2)
  in Ada.Directories.Hierarchical_File_Names A.16.1(10/3)
simple_return_statement 6.5(2/2)
  used 5.1(4/2), P.1
simple_statement 5.1(4/2)
  used 5.1(3), P.1
Sin
  in Ada.Numerics.Generic_Complex_-Elementary_Functions G.1.2(4)
  in Ada.Numerics.Generic_Elementary_-Functions A.5.1(5)
single
  class expected type 8.6(27/2)
single entry 9.5.2(20)
Single_Precision_Complex_Types
  in Interfaces.Fortran B.5(8)
single_protected_declaration 9.4(3/3)
  used 3.3.1(2/3), P.1
single_task_declaration 9.1(3/5)
  used 3.3.1(2/3), P.1
Sinh
  in Ada.Numerics.Generic_Complex_-Elementary_Functions G.1.2(6)
  in Ada.Numerics.Generic_Elementary_-Functions A.5.1(7)
size
  of an object 13.1(7/5)
  in Ada.Direct_IO A.8.4(15)
  in Ada.Directories A.16(26/2), A.16(41/2)
  in Ada.Streams.Stream_IO A.12.1(23)
Size (object) aspect 13.3(41)
Size (subtype) aspect 13.3(48)
Size attribute 13.3(40), 13.3(45)
Size clause 13.3(7/2), 13.3(41), 13.3(48)
size_t
  in Interfaces.C B.3(13)
Skip_Line
  in Ada.Text_IO A.10.1(29/5)
Skip_Page
  in Ada.Text_IO A.10.1(32/5)
slice 4.1.2(2)
  used 4.1(2/5), P.1
  in Ada.Strings.Bounded A.4.4(28)
  in Ada.Strings.Unbounded A.4.5(22)
small
  of a fixed point type 3.5.9(8/2)
Small aspect 3.5.10(2/5)
Small attribute 3.5.10(2/5)
Small clause 3.5.10(2/5), 13.3(7/2)
smallest first order A.18(5/5)
SO
  in Ada.Characters.Latin_1 A.3.3(5)
Soft_Hyphen
  in Ada.Characters.Latin_1 A.3.3(21/3)
SOH
  in Ada.Characters.Latin_1 A.3.3(5)
solidus 2.1(15/5)
  in Ada.Characters.Latin_1 A.3.3(8)
Solve
  in Ada.Numerics.Generic_Complex_-Arrays G.3.2(46/2)
  in Ada.Numerics.Generic_Real_Arrays G.3.1(24/2)
Sort
  in Ada.Containers.Doubly_Linked_-Lists A.18.3(49/5)
  in Ada.Containers.Vectors A.18.2(77/5)
SOS
  in Ada.Characters.Latin_1 A.3.3(19)
SPA
  in Ada.Characters.Latin_1 A.3.3(18)
Space
  in Ada.Characters.Latin_1 A.3.3(8)
  in Ada.Strings A.4.1(4/2)
special file A.16(45/2)
special graphic character
  a category of Character A.3.2(32)
Special_Set
  in Ada.Strings.Maps.Constants A.4.6(4)
specialized needs annex 1.3.3(36/5)
Specialized Needs Annexes 1.1.2(7)
specifiable
  of Address for entries J.7.1(6)
  of Address for stand-alone objects and for program units 13.3(12)
  of Alignment for first subtypes 13.3(26.4/2)
  of Alignment for objects 13.3(25/2)
  of Bit_Order for record types and record extensions 13.5.3(4)
  of Component_Size for array types 13.3(70)
  of External_Tag for a tagged type 13.3(75/3), K.2(65)
  of Input for a type 13.13.2(38/5)

of Machine_Radix for decimal first subtypes F.1(1/5)
of Output for a type 13.13.2(38/5)
of Read for a type 13.13.2(38/5)
of Size for first subtypes 13.3(48)
of Size for stand-alone objects 13.3(41)
of Small for fixed point types 3.5.10(2/5)
of Storage_Pool for a nonderived access-to-object type 13.11(15/5)
of Storage_Size for a nonderived access-to-object type 13.11(15/5)
of Storage_Size for a task first subtype J.9(3/3)
of Write for a type 13.13.2(38/5)
specifiable (of an attribute and for an entity) 13.3(5/3)
specific handler C.7.3(9/2)
specific postcondition expression 6.1.1(4/5)
specific precondition expression 6.1.1(2/5)
specific stable property function 7.3.4(5/5)
specific type 3.4.1(3/5)
Specific_Handler
in Ada.Task_Termination C.7.3(6/2)
specified
of an aspect of representation of an entity 13.1(17)
of an operational aspect of an entity 13.1(18.1/5)
specified (not!) 1.1.3(18), M.2(1.a)
specified as independently addressable C.6(8.1/4)
specified discriminant 3.7(18)
Splice
in Ada.Containers.Doubly_Linked_Lists A.18.3(30/5), A.18.3(31/5), A.18.3(32/2)
Splice_Children
in Ada.Containers.Multiway_Trees A.18.10(57/5), A.18.10(58/5)
Splice_Subtree
in Ada.Containers.Multiway_Trees A.18.10(55/5), A.18.10(56/5)
Split
in Ada.Calendar 9.6(14)
in Ada.Calendar.Formatting 9.6.1(29/2), 9.6.1(32/2), 9.6.1(33/2), 9.6.1(34/2)
in Ada.Execution_Time D.14(8/2)
in Ada.Real_Time D.8(16)
Split_Into_Chunks
in Ada.Iterator_Interfaces 5.5.1(4.4/5)
Sqrt
in Ada.Numerics.Generic_Complex_Elementary_Functions G.1.2(3)
in Ada.Numerics.Generic_Elementary_Functions A.5.1(4)
squirrel away 8.5.4(8.g), 13.9.1(14.e/3)
SS2
in Ada.Characters.Latin_1 A.3.3(17)
SS3
in Ada.Characters.Latin_1 A.3.3(17)
SSA
in Ada.Characters.Latin_1 A.3.3(17)
ST
in Ada.Characters.Latin_1 A.3.3(19)
stabilized view
list A.18.3(151.1/5)
map A.18.4(75.1/5)
set A.18.7(96.13/5)
tree A.18.10(219.1/5)
vector A.18.2(237.1/5)
stable
list A.18.3(151.1/5)
map A.18.4(75.1/5)
set A.18.7(96.13/5)
tree A.18.10(219.1/5)
vector A.18.2(237.1/5)
in Ada.Containers.Doubly_Linked_Lists A.18.3(51.1/5)
in Ada.Containers.Hashed_Maps A.18.5(37.2/5)
in Ada.Containers.Hashed_Sets A.18.8(59.1/5)
in Ada.Containers.Multiway_Trees A.18.10(70.1/5)
in Ada.Containers.Ordered_Maps A.18.6(51.3/5)
in Ada.Containers.Ordered_Sets A.18.9(74.1/5)
in Ada.Containers.Vectors A.18.2(79.1/5)
stable property 1.3.1(73/5), 7.3.4(1/5)
stable property function
class-wide 7.3.4(6/5)
of a type 7.3.4(7/5)
specific 7.3.4(5/5)
Stable_Properties aspect 7.3.4(5/5)
Stable_Properties'Class aspect 7.3.4(6/5)
stand-alone constant 3.3.1(23/5)
corresponding to a formal object of mode in 12.4(10/2)
stand-alone object 3.3.1(1/3)
[*partial*] 12.4(10/2)
stand-alone variable 3.3.1(23/5)
Standard A.1(4/5)
standard error file A.10(6)
standard input file A.10(5)
standard mode 1.1.5(11)
standard output file A.10(5)
standard storage pool 13.11(17)
Standard_Error
in Ada.Text_IO A.10.1(16), A.10.1(19)
Standard_Indent
in Ada.Strings.Text_Buffers A.4.12(13/5)
Standard_Input
in Ada.Text_IO A.10.1(16), A.10.1(19)
Standard_Output
in Ada.Text_IO A.10.1(16), A.10.1(19)
Start_Search
in Ada.Directories A.16(32/2)
State
in Ada.Numerics.Discrete_Random A.5.2(23)
in Ada.Numerics.Float_Random A.5.2(11)
statement 5.1(3)
used 5.1(2/3), P.1
statement_identifier 5.1(8)
used 5.1(7), 5.5(2), 5.6(2), P.1
static 3.3.2(1.a/5), 4.9(1)
constant 4.9(24/5)
constraint 4.9(27)
delta constraint 4.9(29)
digits constraint 4.9(29)
discrete_range 4.9(25)
discriminant constraint 4.9(31)
expression 4.9(2)
function 4.9(18)
index constraint 4.9(30)
range 4.9(25)
range constraint 4.9(29)
scalar subtype 4.9(26/3)
string subtype 4.9(26/3)
subtype 4.9(26/3)
subtype 12.4(9/2)
value 4.9(13.a)
Static aspect 6.8(3.6/5)
static expression function 6.8(3.6/5)
static semantics 1.1.2(28)
Static_Predicate aspect 3.2.4(1/5)
statically compatible
for a constraint and a scalar subtype 4.9.1(4/3)
for a constraint and an access or composite subtype 4.9.1(4/3)
for two subtypes 4.9.1(5/3)
statically constrained 4.9(32)
statically deeper 3.10.2(4), 3.10.2(17)
statically denote 4.9(14)
statically determined tag 3.9.2(1/5)
[*partial*] 3.9.2(15), 3.9.2(19)
statically match
for global aspects 4.9.1(1.5/5)
statically matching
effect on subtype-specific aspects 13.1(14/5)
for constraints 4.9.1(1/2)
for ranges 4.9.1(3)
for subtypes 4.9.1(2/5)
required 3.9.2(10/2), 3.10.2(27.1/2), 4.6(24.15/2), 4.6(24.5/2), 6.3.1(16.3/3), 6.3.1(17/3), 6.3.1(23), 6.5(5.3/5), 7.3(13), 8.5.1(4.2/2), 12.4(8.1/2), 12.5.1(14), 12.5.3(6), 12.5.3(7), 12.5.4(3), 12.7(7)
Statically names 4.9(17.1/5)
statically tagged 3.9.2(4/2)
statically unevaluated 4.9(32.1/3)
Status_Error
in Ada.Direct_IO A.8.4(18)
in Ada.Directories A.16(43/2)
in Ada.IO_Exceptions A.13(4)
in Ada.Sequential_IO A.8.1(15)
in Ada.Streams.Stream_IO A.12.1(26)

in Ada.Text_IO A.10.1(85/5)
Storage
child of Ada.Streams 13.13.1(12/5)
storage deallocation
unchecked 13.11.2(1)
storage element 13.3(8)
storage management
user-defined 13.11(1)
storage node E(2)
storage place
of a component 13.5(1)
representation aspect 13.5(1)
storage place attributes
of a component 13.5.2(1)
storage pool 3.10(7/1)
default 13.11.3(4.2/4)
storage pool element 13.11(11)
storage pool object 1.3.1(74/5)
storage pool that supports subpools 13.11.4(18/3)
storage pool type 13.11(11)
Storage_Array
in System.Storage_Elements 13.7.1(5)
Storage_Check 11.5(23)
[*partial*] 11.1(6/5), 13.3(67), 13.11(17), D.7(17/1), D.7(18/1), D.7(19/1)
Storage_Count *subtype of* Storage_Offset
in System.Storage_Elements 13.7.1(4)
Storage_Element
in System.Storage_Elements 13.7.1(5)
Storage_Elements
child of System 13.7.1(2/5)
Storage_Error
raised by detection of a bounded error 8.5.4(8.1/1)
raised by failure of runtime check 11.1(4), 11.1(6/5), 11.5(23), 13.3(67), 13.11(17), 13.11(18/5), D.7(17/1), D.7(18/1), D.7(19.3/3), D.7(19/1)
in Standard A.1(46)
Storage_IO
child of Ada A.9(3/5)
Storage_Offset
in System.Storage_Elements 13.7.1(3)
Storage_Pool aspect 13.11(15/5)
Storage_Pool attribute 13.11(13)
Storage_Pool clause 13.3(7/2), 13.11(15/5)
storage_pool_indicator 13.11.3(3.1/4)
used 13.11.3(3/3), L(8.3/3)
Storage_Pools
child of System 13.11(5/5)
Storage_Size
in System.Storage_Pools 13.11(9)
in System.Storage_Pools.Subpools 13.11.4(16/3)
Storage_Size (access) aspect 13.11(15/5)
Storage_Size (task) aspect 13.3(65.2/3)
Storage_Size attribute 13.3(60/3), 13.11(14), J.9(2)
Storage_Size clause 13.3(7/2), 13.11(15/5)
Storage_Size pragma J.15.4(2/3), L(35.1/3)
Storage_Stream_Type
in Ada.Streams.Storage 13.13.1(13/5)
Storage_Unit
in System 13.7(13)
stream 1.3.1(76/5), 13.13(1)
in Ada.Streams.Stream_IO A.12.1(13)
in Ada.Text_IO.Text_Streams A.12.2(4)
in Ada.Wide_Text_IO.Text_Streams A.12.3(4)
in Ada.Wide_Wide_Text_IO.Text_-Streams A.12.4(4/2)
stream file A.8(1/2)
stream type 13.13(1)
stream-oriented attributes 13.13.2(1/3)
Stream_Access
in Ada.Streams.Stream_IO A.12.1(4)
in Ada.Text_IO.Text_Streams A.12.2(3/5)
in Ada.Wide_Text_IO.Text_Streams A.12.3(3/5)
in Ada.Wide_Wide_Text_IO.Text_-Streams A.12.4(3/5)
Stream_Element
in Ada.Streams 13.13.1(4/1)
Stream_Element_Array
in Ada.Streams 13.13.1(4/1)
Stream_Element_Count *subtype of* Stream_Element_Offset
in Ada.Streams 13.13.1(4/1)
Stream_Element_Offset
in Ada.Streams 13.13.1(4/1)
Stream_IO
child of Ada.Streams A.12.1(3/5)
Stream_Size aspect 13.13.2(1.5/2)
Stream_Size attribute 13.13.2(1.2/3)
Stream_Size clause 13.3(7/2)
Stream_Type
in Ada.Streams.Storage.Bounded 13.13.1(27/5)
in Ada.Streams.Storage.Unbounded 13.13.1(19/5)
Streams
child of Ada 13.13.1(2/5)
strict mode G.2(1)
strict weak ordering A.18(5/5)
String
in Standard A.1(37/3)
string type 3.6.3(1)
String_Access
in Ada.Strings.Unbounded A.4.5(7)
string_element 2.6(3)
used 2.6(2), P.1
string_literal 2.6(2)
used 4.4(7/5), 6.1(9), P.1
String_Literal aspect 4.2.1(5/5)
Strings
child of Ada A.4.1(3/5)
child of Ada.Strings.UTF_Encoding A.4.11(22/5)
child of Interfaces.C B.3.1(3/5)
Strings_Assertion_Check 11.5(23.10/5)
Strlen
in Interfaces.C.Strings B.3.1(17)
structure
See record type 3.8(1)
STS
in Ada.Characters.Latin_1 A.3.3(18)
STX
in Ada.Characters.Latin_1 A.3.3(5)
SUB
in Ada.Characters.Latin_1 A.3.3(6)
Sub_Second
in Ada.Calendar.Formatting 9.6.1(27/2)
subaggregate
of an array_aggregate 4.3.3(6)
subcomponent 3.2(6/2)
of the nominal subtype 3.9.1(4.2/5)
subpool 13.11.4(18/3)
subpool access type 13.11.4(22/3)
subpool handle 13.11.4(18/3)
Subpool_Handle
in System.Storage_Pools.Subpools 13.11.4(6/3)
subpool_specification 4.8(2.1/3)
used 4.8(2/3), P.1
Subpools
child of System.Storage_Pools 13.11.4(3/5)
subprogram 1.3.2(7/5), 6(1)
abstract 3.9.3(3/2)
allows exit 5.5.3(19/5)
subprogram call 6.4(1)
subprogram instance 12.3(13)
subprogram property aspect definition 7.3.4(3/5)
subprogram_body 6.3(2/3)
used 3.11(6), 9.4(8/4), 10.1.1(7), P.1
subprogram_body_stub 10.1.3(3/3)
used 10.1.3(2), P.1
subprogram_declaration 6.1(2/3)
used 3.1(3/3), 9.4(5/1), 9.4(8/4), 10.1.1(5), P.1
subprogram_default 12.6(3/2)
used 12.6(2.1/3), 12.6(2.2/3), P.1
subprogram_renaming_declaration 8.5.4(2/3)
used 8.5(2), 10.1.1(6), P.1
subprogram_specification 6.1(4/2)
used 3.9.3(1.1/3), 6.1(2/3), 6.3(2/3), 8.5.4(2/3), 10.1.3(3/3), 12.1(3/3), 12.6(2.1/3), 12.6(2.2/3), P.1
subsequence
reduction attribute 4.5.10(21/5)
subsystem 10.1(3)
subtree
node which roots A.18.10(3/4)
of a tree A.18.10(3/4)
Subtree_Node_Count
in Ada.Containers.Multiway_Trees A.18.10(18.1/5), A.18.10(18/5)
subtype 1.3.1(77/5), 3.2(8/5)
constraint of 3.2(8/5)
of a generic formal object 12.4(10.c)
type of 3.2(8/5)
values belonging to 3.2(8/5)
values outside of 3.2(8/5)

subtype (of an object)
See actual subtype of an object 3.3(23/5)
See actual subtype of an object 3.3.1(9/2)
subtype conformance 6.3.1(17/3), 12.3(11.j/3)
[*partial*] 3.10.2(34/2), 9.5.4(17/5)
required 3.9.2(10/2), 3.10.2(32/5), 4.6(24.20/3), 8.5.1(4.3/2), 8.5.4(5/3), 9.1(9.7/2), 9.1(9.8/2), 9.4(11.6/2), 9.4(11.7/2), 9.5.4(5/3), 12.4(8.2/2), 12.5.4(5/3)
subtype conversion
bounds of a decimal fixed point type 3.5.9(16.a.1/1)
bounds of a fixed point type 3.5.9(14.a.1/1)
bounds of a floating point type 3.5.7(11.a.1/1)
bounds of signed integer type 3.5.4(9.a.1/1)
See type conversion 4.6(1/3)
See also implicit subtype conversion 4.6(1/3)
subtype-specific
attribute_definition_clause 13.3(7.a)
of a representation item 13.1(8/5)
of an aspect 13.1(8/5)
subtype_declaration 3.2.2(2/3)
used 3.1(3/3), P.1
subtype_indication 3.2.2(3/2)
used 3.2.2(2/3), 3.3.1(2/3), 3.4(2/2), 3.6(6), 3.6(7/2), 3.6.1(3), 3.8.1(5/3), 3.10(3), 4.8(2/3), 5.5.2(2.1/5), 6.5(2.3/2), 7.3(3/3), P.1
subtype_mark 3.2.2(4)
used 3.2.2(3/2), 3.6(4), 3.7(5/5), 3.9.4(3/2), 3.10(6/2), 4.3.2(3), 4.4(3.2/4), 4.6(2), 4.7(2), 6.1(13/2), 6.1(15/5), 8.4(4/3), 8.5.1(2/5), 12.3(5), 12.4(2/3), 12.5(2.1/5), 12.5(2.2/5), 12.5.1(3/5), 13.8(14.b), H.7.1(6/5), P.1
subtypes
of a profile 6.1(25)
subunit 1.3.3(37/5), 10.1.3(7), 10.1.3(8/2)
of a program unit 10.1.3(8/2)
used 10.1.1(3), P.1
Succ attribute 3.5(22)
Success
in Ada.Command_Line A.15(8)
successor element
of a hashed set A.18.8(68/2)
of a set A.18.7(6/2)
of an ordered set A.18.9(81/3)
successor node
of a hashed map A.18.5(46/2)
of a map A.18.4(6/2)
of an ordered map A.18.6(58/3)
Sunday
in Ada.Calendar.Formatting 9.6.1(17/2)
super
See view conversion 4.6(5/5)
Superscript_One
in Ada.Characters.Latin_1 A.3.3(22)
Superscript_Three
in Ada.Characters.Latin_1 A.3.3(22)
Superscript_Two
in Ada.Characters.Latin_1 A.3.3(22)
support external streaming 13.13.2(52/3)
Supported
in Ada.Execution_Time.Interrupts D.14.3(3/5)
suppress a check 1.3.5(6/5)
Suppress pragma 11.5(4/2), J.10(3/2), L(36)
suppressed check 11.5(8/2)
Suspend_Until_True
in Ada.Synchronous_Task_Control D.10(4/5)
Suspend_Until_True_And_Set_Deadline
in Ada.Synchronous_Task_Control.EDF D.10(5.2/5)
Suspension_Object
in Ada.Synchronous_Task_Control D.10(4/5)
Swap
in Ada.Containers.Doubly_Linked_Lists A.18.3(28/5)
in Ada.Containers.Indefinite_Holders A.18.18(22.1/5)
in Ada.Containers.Multiway_Trees A.18.10(37/5)
in Ada.Containers.Vectors A.18.2(55/5), A.18.2(56/5)
Swap_Links
in Ada.Containers.Doubly_Linked_Lists A.18.3(29/5)
Symmetric_Difference
in Ada.Containers.Hashed_Sets A.18.8(35/5), A.18.8(36/5)
in Ada.Containers.Ordered_Sets A.18.9(36/5), A.18.9(37/5)
SYN
in Ada.Characters.Latin_1 A.3.3(6)
synchronization 9(1/5)
Synchronization aspect 9.5(12/3)
synchronization_kind 9.5(10/3)
synchronized 1.3.1(78/5)
synchronized interface 3.9.4(5/2)
synchronized object 9.10(18/5)
synchronized tagged type 3.9.4(6/2)
Synchronized_Queue_Interfaces
child of Ada.Containers A.18.27(3/5)
Synchronous_Barrier
in Ada.Synchronous_Barriers D.10.1(5/3)
Synchronous_Barriers
child of Ada D.10.1(3/5)
Synchronous_Task_Control
child of Ada D.10(3/5)
syntactic category 1.1.4(15)
syntax
complete listing P.1(1/5)
cross reference P.2(1/3)
notation 1.1.4(3)
under Syntax heading 1.1.2(25)
System 13.7(3/5)
System.Address_To_Access_Conversions 13.7.2(2/5)
System.Atomic_Operations C.6.1(3/5)
System.Atomic_Operations.Exchange C.6.2(5/5)
System.Atomic_Operations.Integer_Arithmetic C.6.4(3/5)
System.Atomic_Operations.Modular_Arithmetic C.6.5(3/5)
System.Atomic_Operations.Test_And_Set C.6.3(3/5)
System.Machine_Code 13.8(7)
System.Multiprocessors D.16(3/5)
System.Multiprocessors.Dispatching_Domains D.16.1(3/5)
System.RPC E.5(3/5)
System.Storage_Elements 13.7.1(2/5)
System.Storage_Pools 13.11(5/5)
System.Storage_Pools.Subpools 13.11.4(3/5)
System_Assertion_Check 11.5(23.11/5)
System_Dispatching_Domain
in System.Multiprocessors.Dispatching_Domains D.16.1(6/3)
System_Name
in System 13.7(4)
systems programming C(1)

T

T
italicized 4.5.1(3.b/2)
Tag
in Ada.Tags 3.9(6/5)
Tag attribute 3.9(16), 3.9(18)
tag indeterminate 3.9.2(6/2)
tag of an object 3.9(3)
class-wide object 3.9(22)
object created by an allocator 3.9(21/5)
preserved by type conversion and parameter passing 3.9(25)
returned by a function 3.9(23), 3.9(24/2)
stand-alone object, component, or aggregate 3.9(20)
Tag_Array
in Ada.Tags 3.9(7.3/2)
Tag_Check 11.5(18)
[*partial*] 3.9.2(16), 4.6(42), 4.6(52), 5.2(10), 6.5(8.2/5)
Tag_Error
in Ada.Tags 3.9(8)
tagged incomplete view 3.10.1(2.1/4)
tagged type 1.3.1(80/5), 3.9(2/2)
protected 3.9.4(6/2)
synchronized 3.9.4(6/2)
task 3.9.4(6/2)
Tags
child of Ada 3.9(6/5)

Tail
  in Ada.Strings.Bounded A.4.4(72), A.4.4(73)
  in Ada.Strings.Fixed A.4.3(37), A.4.3(38)
  in Ada.Strings.Unbounded A.4.5(67), A.4.5(68)
tail (of a queue) D.2.1(5/2)
tamper with cursors
  [*partial*] A.18(2.1/5)
  of a list A.18.3(61/5)
  of a map A.18.4(7/5)
  of a set A.18.7(7/5)
  of a tree A.18.10(80/5)
  of a vector A.18.2(90/5)
tamper with elements
  [*partial*] A.18(2.1/5)
  of a holder A.18.18(29/5)
  of a list A.18.3(61/5)
  of a map A.18.4(7/5)
  of a set A.18.7(7/5)
  of a tree A.18.10(80/5)
  of a vector A.18.2(90/5)
tampering
  prohibited for a holder A.18.18(29/5)
  prohibited for a list A.18.3(61/5)
  prohibited for a map A.18.4(7/5)
  prohibited for a set A.18.7(7/5)
  prohibited for a tree A.18.10(80/5)
  prohibited for a vector A.18.2(90/5)
Tampering_With_Cursors_Prohibited
  in Ada.Containers.Doubly_Linked_-Lists A.18.3(10.1/5)
  in Ada.Containers.Hashed_Maps A.18.5(7.1/5)
  in Ada.Containers.Hashed_Sets A.18.8(8.1/5)
  in Ada.Containers.Multiway_Trees A.18.10(15.1/5)
  in Ada.Containers.Ordered_Maps A.18.6(8.1/5)
  in Ada.Containers.Ordered_Sets A.18.9(9.1/5)
  in Ada.Containers.Vectors A.18.2(12.1/5)
Tampering_With_Elements_Prohibited
  in Ada.Containers.Doubly_Linked_-Lists A.18.3(10.2/5)
  in Ada.Containers.Hashed_Maps A.18.5(7.2/5)
  in Ada.Containers.Multiway_Trees A.18.10(15.2/5)
  in Ada.Containers.Ordered_Maps A.18.6(8.2/5)
  in Ada.Containers.Vectors A.18.2(12.2/5)
Tampering_With_The_Element_Prohibited
  in Ada.Containers.Indefinite_Holders A.18.18(8.1/5)
Tan
  in Ada.Numerics.Generic_Complex_-Elementary_Functions G.1.2(4)
  in Ada.Numerics.Generic_Elementary_-Functions A.5.1(5)
Tanh
  in Ada.Numerics.Generic_Complex_-Elementary_Functions G.1.2(6)
  in Ada.Numerics.Generic_Elementary_-Functions A.5.1(7)
target
  of an assignment operation 5.2(3)
  of an assignment_statement 5.2(3)
target name 5.2.1(1/5)
target object
  of a requeue_statement 9.5(7)
  of the name of an entry or a protected subprogram 9.5(2/3)
target statement
  of a goto_statement 5.8(3)
target subtype
  of a type_conversion 4.6(3)
target_name 5.2.1(2/5)
  used 4.1(2/5), P.1
task 9(1/5)
  activation 9.2(1)
  completion 9.3(1)
  dependence 9.3(1)
  execution 9.2(1)
  termination 9.3(1)
task declaration 9.1(1)
task dispatching D.2.1(4/5)
task dispatching point D.2.1(4/5)
  [*partial*] D.2.3(8/2), D.2.4(9/3)
task dispatching policy 9(10.a/3), D.2.2(6/2)
  [*partial*] D.2.1(5/2)
  EDF_Within_Priorities D.2.6(7/5)
  FIFO_Within_Priorities D.2.3(2/2)
  Non_Preemptive_FIFO_Within_-Priorities D.2.4(2/2)
  Round_Robin_Within_Priorities D.2.5(2/2)
task interface 3.9.4(5/2)
task priority D.1(15/5)
task state
  abnormal 9.8(4)
  blocked 9(10/5)
  callable 9.9(2)
  held D.11(4/2)
  inactive 9(10/5)
  ready 9(10/5)
  terminated 9(10/5)
task tagged type 3.9.4(6/2)
task type 1.3.1(82/5)
task unit 9(9)
Task_Array
  in Ada.Execution_Time.Group_Budgets D.14.2(6/2)
Task_Attributes
  child of Ada C.7.2(2/5)
task_body 9.1(6/3)
  used 3.11(6), P.1
task_body_stub 10.1.3(5/3)
  used 10.1.3(2), P.1
task_definition 9.1(4)
  used 9.1(2/3), 9.1(3/5), P.1
Task_Dispatching_Policy pragma D.2.2(2), L(37)
Task_Id
  in Ada.Task_Identification C.7.1(2/5)
Task_Identification
  child of Ada C.7.1(2/5)
task_item 9.1(5/1)
  used 9.1(4), P.1
Task_Termination
  child of Ada C.7.3(2/5)
task_type_declaration 9.1(2/3)
  used 3.2.1(3/3), P.1
Tasking_Check 11.5(23.2/5)
  [*partial*] 9.2(5), 9.5.3(21)
Tasking_Error
  raised by detection of a bounded error 13.11.2(13)
  raised by failure of runtime check 9.2(5), 9.5.3(21), 11.1(4), 11.5(23.1/5)
  in Standard A.1(46)
template 12(1)
  for a formal package 12.7(4)
  See generic unit 12(1)
term 4.4(5)
  used 4.4(4), P.1
terminal interrupt
  example 9.7.4(10)
terminate_alternative 9.7.1(7)
  used 9.7.1(4), P.1
terminated
  a task state 9(10/5)
Terminated attribute 9.9(3)
termination
  abnormal 10.2(25.c)
  normal 10.2(25.c)
  of a partition 10.2(25.c)
  of a partition E.1(7)
termination handler C.7.3(8/3)
  fall-back C.7.3(9/2)
  specific C.7.3(9/2)
Termination_Handler
  in Ada.Task_Termination C.7.3(4/2)
Terminator_Error
  in Interfaces.C B.3(40)
Test_And_Set
  child of System.Atomic_Operations C.6.3(3/5)
Test_And_Set_Flag
  in System.Atomic_Operations.Test_And_Set C.6.3(4/5)
tested type
  of a membership test 4.5.2(3/3)
text of a program 2.2(1)
Text_Buffer_Count
  in Ada.Strings.Text_Buffers A.4.12(4/5)
Text_Buffers
  child of Ada.Strings A.4.12(3/5)

Text_IO
  child of Ada A.10.1(2/5)
Text_Streams
  child of Ada.Text_IO A.12.2(3/5)
  child of Ada.Wide_Text_IO A.12.3(3/5)
  child of Ada.Wide_Wide_Text_IO A.12.4(3/5)
Text_Truncated
  in Ada.Strings.Text_Buffers.Bounded A.4.12(28/5)
thread
  logical 9(1/5)
throw (an exception)
  See raise 11(1/3)
thunk 13.14(19.i)
Thursday
  in Ada.Calendar.Formatting 9.6.1(17/2)
tick 2.1(15/5)
  in Ada.Real_Time D.8(6)
  in System 13.7(10)
tied
  accessibility levels 3.10.2(5/5)
Tilde
  in Ada.Characters.Latin_1 A.3.3(14)
Time
  in Ada.Calendar 9.6(10/5)
  in Ada.Real_Time D.8(4)
time base 9.6(6/3)
time limit
  example 9.7.4(12)
time type 9.6(6/3)
Time-dependent Reset procedure
  of the random number generator A.5.2(34)
time-out
  example 9.7.4(12)
  See asynchronous_select 9.7.4(12)
  See selective_accept 9.7.1(1)
  See timed_entry_call 9.7.2(1/5)
Time_Error
  in Ada.Calendar 9.6(18)
Time_First
  in Ada.Real_Time D.8(4)
Time_Last
  in Ada.Real_Time D.8(4)
Time_Of
  in Ada.Calendar 9.6(15)
  in Ada.Calendar.Formatting 9.6.1(30/2), 9.6.1(31/2)
  in Ada.Execution_Time D.14(9/2)
  in Ada.Real_Time D.8(16)
Time_Of_Event
  in Ada.Real_Time.Timing_Events D.15(6/2)
Time_Offset
  in Ada.Calendar.Time_Zones 9.6.1(4/2)
Time_Remaining
  in Ada.Execution_Time.Timers D.14.1(8/2)
Time_Span
  in Ada.Real_Time D.8(5)
Time_Span_First
  in Ada.Real_Time D.8(5)
Time_Span_Last
  in Ada.Real_Time D.8(5)
Time_Span_Unit
  in Ada.Real_Time D.8(5)
Time_Span_Zero
  in Ada.Real_Time D.8(5)
Time_Unit
  in Ada.Real_Time D.8(4)
Time_Zones
  child of Ada.Calendar 9.6.1(2/5)
timed_entry_call 9.7.2(2)
  used 9.7(2), P.1
Timer
  in Ada.Execution_Time.Timers D.14.1(4/2)
timer interrupt
  example 9.7.4(12)
Timer_Handler
  in Ada.Execution_Time.Timers D.14.1(5/5)
Timer_Resource_Error
  in Ada.Execution_Time.Timers D.14.1(9/2)
Timers
  child of Ada.Execution_Time D.14.1(3/5)
times operator 4.5(1), 4.5.5(1)
timing
  See delay_statement 9.6(1)
Timing_Event
  in Ada.Real_Time.Timing_Events D.15(4/5)
Timing_Event_Handler
  in Ada.Real_Time.Timing_Events D.15(4/5)
Timing_Events
  child of Ada.Real_Time D.15(3/5)
To_Ada
  in Interfaces.C B.3(22), B.3(26), B.3(28), B.3(32), B.3(37), B.3(39), B.3(39.10/2), B.3(39.13/2), B.3(39.17/2), B.3(39.19/2), B.3(39.4/2), B.3(39.8/2)
  in Interfaces.COBOL B.4(17), B.4(19)
  in Interfaces.Fortran B.5(13), B.5(14), B.5(16)
To_Address
  in System.Address_To_Access_-Conversions 13.7.2(3/3)
  in System.Storage_Elements 13.7.1(10/3)
To_Basic
  in Ada.Characters.Handling A.3.2(6), A.3.2(7)
  in Ada.Wide_Characters.Handling A.3.5(20.1/5), A.3.5(21.1/5)
To_Big_Integer
  in Ada.Numerics.Big_Numbers.Big_Integers A.5.6(7/5), A.5.6(12/5), A.5.6(13/5)
To_Big_Real
  in Ada.Numerics.Big_Numbers.Big_Reals A.5.7(9/5), A.5.7(13/5), A.5.7(14/5)
To_Binary
  in Interfaces.COBOL B.4(45), B.4(48)
To_Bounded_String
  in Ada.Strings.Bounded A.4.4(11)
To_C
  in Interfaces.C B.3(21), B.3(25), B.3(27), B.3(32), B.3(36), B.3(38), B.3(39.13/2), B.3(39.16/2), B.3(39.18/2), B.3(39.4/2), B.3(39.7/2), B.3(39.9/2)
To_Character
  in Ada.Characters.Conversions A.3.4(5/2)
To_Chars_Ptr
  in Interfaces.C.Strings B.3.1(8)
To_COBOL
  in Interfaces.COBOL B.4(17), B.4(18)
To_Cursor
  in Ada.Containers.Vectors A.18.2(25/5)
To_Decimal
  in Interfaces.COBOL B.4(35), B.4(40), B.4(44), B.4(47)
To_Display
  in Interfaces.COBOL B.4(36)
To_Domain
  in Ada.Strings.Maps A.4.2(24)
  in Ada.Strings.Wide_Maps A.4.7(24)
  in Ada.Strings.Wide_Wide_Maps A.4.8(24/2)
To_Duration
  in Ada.Real_Time D.8(13)
To_Fortran
  in Interfaces.Fortran B.5(13), B.5(14), B.5(15)
To_Holder
  in Ada.Containers.Indefinite_Holders A.18.18(9/5)
To_Index
  in Ada.Containers.Vectors A.18.2(26.1/5), A.18.2(26/5)
To_Integer
  in Ada.Numerics.Big_Numbers.Big_Integers A.5.6(11/5)
  in System.Storage_Elements 13.7.1(10/3)
To_ISO_646
  in Ada.Characters.Handling A.3.2(11), A.3.2(12)
To_Long_Binary
  in Interfaces.COBOL B.4(48)
To_Lower
  in Ada.Characters.Handling A.3.2(6), A.3.2(7)
  in Ada.Wide_Characters.Handling A.3.5(20/3), A.3.5(21/3)
To_Mapping
  in Ada.Strings.Maps A.4.2(23)
  in Ada.Strings.Wide_Maps A.4.7(23)

in Ada.Strings.Wide_Wide_Maps A.4.8(23/2)
To_Packed
in Interfaces.COBOL B.4(41)
To_Picture
in Ada.Text_IO.Editing F.3.3(6)
To_Pointer
in System.Address_To_Access_-Conversions 13.7.2(3/3)
To_Quotient_String
in Ada.Numerics.Big_Numbers.Big_Reals A.5.7(19/5)
To_Range
in Ada.Strings.Maps A.4.2(24)
in Ada.Strings.Wide_Maps A.4.7(25)
in Ada.Strings.Wide_Wide_Maps A.4.8(25/2)
To_Ranges
in Ada.Strings.Maps A.4.2(10)
in Ada.Strings.Wide_Maps A.4.7(10)
in Ada.Strings.Wide_Wide_Maps A.4.8(10/2)
To_Real
in Ada.Numerics.Big_Numbers.Big_Reals A.5.7(10/5)
To_Sequence
in Ada.Strings.Maps A.4.2(19)
in Ada.Strings.Wide_Maps A.4.7(19)
in Ada.Strings.Wide_Wide_Maps A.4.8(19/2)
To_Set
in Ada.Containers.Hashed_Sets A.18.8(9/5)
in Ada.Containers.Ordered_Sets A.18.9(10/5)
in Ada.Strings.Maps A.4.2(8), A.4.2(9), A.4.2(17), A.4.2(18)
in Ada.Strings.Wide_Maps A.4.7(8), A.4.7(9), A.4.7(17), A.4.7(18)
in Ada.Strings.Wide_Wide_Maps A.4.8(8/2), A.4.8(9/2), A.4.8(17/2), A.4.8(18/2)
To_String
in Ada.Characters.Conversions A.3.4(5/2)
in Ada.Numerics.Big_Numbers.Big_Integers A.5.6(14/5)
in Ada.Numerics.Big_Numbers.Big_Reals A.5.7(15/5)
in Ada.Strings.Bounded A.4.4(12)
in Ada.Strings.Unbounded A.4.5(11)
To_Time_Span
in Ada.Real_Time D.8(13)
To_Unbounded_String
in Ada.Strings.Unbounded A.4.5(9), A.4.5(10)
To_Upper
in Ada.Characters.Handling A.3.2(6), A.3.2(7)
in Ada.Wide_Characters.Handling A.3.5(20/3), A.3.5(21/3)
To_Vector
in Ada.Containers.Vectors A.18.2(13/5), A.18.2(14/5)
To_Wide_Character
in Ada.Characters.Conversions A.3.4(4/2), A.3.4(5/2)
To_Wide_String
in Ada.Characters.Conversions A.3.4(4/2), A.3.4(5/2)
To_Wide_Wide_Character
in Ada.Characters.Conversions A.3.4(4/2)
To_Wide_Wide_String
in Ada.Characters.Conversions A.3.4(4/2)
token
See lexical element 2.2(1)
Trailing_Nonseparate
in Interfaces.COBOL B.4(23)
Trailing_Separate
in Interfaces.COBOL B.4(23)
transfer of control 5.1(14/2)
Translate
in Ada.Strings.Bounded A.4.4(53), A.4.4(54), A.4.4(55), A.4.4(56)
in Ada.Strings.Fixed A.4.3(18), A.4.3(19), A.4.3(20), A.4.3(21)
in Ada.Strings.Unbounded A.4.5(48), A.4.5(49), A.4.5(50), A.4.5(51)
Translation_Error
in Ada.Strings A.4.1(5)
Transpose
in Ada.Numerics.Generic_Complex_-Arrays G.3.2(34/2)
in Ada.Numerics.Generic_Real_Arrays G.3.1(17/2)
Tree
in Ada.Containers.Multiway_Trees A.18.10(8/5), A.18.10(70.2/5)
Tree_Iterator_Interfaces
in Ada.Containers.Multiway_Trees A.18.10(13/3), A.18.10(70.7/5)
triggering_alternative 9.7.4(3)
used 9.7.4(2), P.1
triggering_statement 9.7.4(4/2)
used 9.7.4(3), P.1
Trim
in Ada.Strings.Bounded A.4.4(67), A.4.4(68), A.4.4(69)
in Ada.Strings.Fixed A.4.3(31), A.4.3(32), A.4.3(33), A.4.3(34)
in Ada.Strings.Unbounded A.4.5(61), A.4.5(62), A.4.5(63), A.4.5(64)
Trim_End
in Ada.Strings A.4.1(6)
True 3.5.3(1)
Truncation
in Ada.Strings A.4.1(6)
Truncation attribute A.5.3(42)
Tuesday
in Ada.Calendar.Formatting 9.6.1(17/2)
two's complement
modular types 3.5.4(29)
type 1.3.1(84/5), 3.2(1)
abstract 3.9.3(1.2/2)
needs finalization 7.6(9.1/2)
of a subtype 3.2(8/5)
stable property function of 7.3.4(7/5)
synchronized tagged 3.9.4(6/2)
See also tag 3.9(3)
See also language-defined types
type conformance 6.3.1(15/2)
[*partial*] 3.4(17/2), 8.3(8), 8.3(26/2), 10.1.4(4/3)
required 3.11.1(5), 4.1.4(17/5), 8.6(26), 9.1(9.2/3), 9.1(9.5/3), 9.4(11.1/3), 9.4(11.4/3), 9.5.4(3/3), 12.4(5/2)
type conversion 4.6(1/3)
access 4.6(24.11/2), 4.6(24.18/2), 4.6(24.19/2), 4.6(47)
arbitrary order 1.1.4(18)
array 4.6(24.2/2), 4.6(36)
composite (non-array) 4.6(21/3), 4.6(40)
enumeration 4.6(21.1/2), 4.6(34)
numeric 4.6(24.1/2), 4.6(29)
unchecked 13.9(1)
See also qualified_expression 4.7(1)
type conversion, implicit
See implicit subtype conversion 4.6(1/3)
type extension 3.9(2/2), 3.9.1(1/2)
type invariant 7.3.2(2/3)
class-wide 7.3.2(3/5)
See also invariant
type of a discrete_range 3.6.1(4)
type of a range 3.5(4)
type parameter
See discriminant 3.7(1/2)
type profile
See profile, type conformant 6.3.1(15/2)
type property aspect definition 7.3.4(3/5)
type resolution rules 8.6(20/2)
if any type in a specified class of types is expected 8.6(21)
if expected type is specific 8.6(22)
if expected type is universal or class-wide 8.6(21)
type tag
See tag 3.9(3)
type-related
aspect 13.1(8.1/3), 13.1(8/5)
attribute_definition_clause 13.3(7.a)
operational item 13.1(8.1/3)
representation item 13.1(8/5)
type_conversion 4.6(2)
used 4.1(2/5), P.1
See also unchecked type conversion 13.9(1)
type_declaration 3.2.1(2)
used 3.1(3/3), P.1
type_definition 3.2.1(4/2)

used 3.2.1(3/3), P.1
Type_Invariant aspect 7.3.2(2/3)
Type_Invariant'Class aspect 7.3.2(3/5)
Type_Set
in Ada.Text_IO A.10.1(7)
types
of a profile 6.1(29)

U

UC_A_Acute
in Ada.Characters.Latin_1 A.3.3(23)
UC_A_Circumflex
in Ada.Characters.Latin_1 A.3.3(23)
UC_A_Diaeresis
in Ada.Characters.Latin_1 A.3.3(23)
UC_A_Grave
in Ada.Characters.Latin_1 A.3.3(23)
UC_A_Ring
in Ada.Characters.Latin_1 A.3.3(23)
UC_A_Tilde
in Ada.Characters.Latin_1 A.3.3(23)
UC_AE_Diphthong
in Ada.Characters.Latin_1 A.3.3(23)
UC_C_Cedilla
in Ada.Characters.Latin_1 A.3.3(23)
UC_E_Acute
in Ada.Characters.Latin_1 A.3.3(23)
UC_E_Circumflex
in Ada.Characters.Latin_1 A.3.3(23)
UC_E_Diaeresis
in Ada.Characters.Latin_1 A.3.3(23)
UC_E_Grave
in Ada.Characters.Latin_1 A.3.3(23)
UC_I_Acute
in Ada.Characters.Latin_1 A.3.3(23)
UC_I_Circumflex
in Ada.Characters.Latin_1 A.3.3(23)
UC_I_Diaeresis
in Ada.Characters.Latin_1 A.3.3(23)
UC_I_Grave
in Ada.Characters.Latin_1 A.3.3(23)
UC_Icelandic_Eth
in Ada.Characters.Latin_1 A.3.3(24)
UC_Icelandic_Thorn
in Ada.Characters.Latin_1 A.3.3(24)
UC_N_Tilde
in Ada.Characters.Latin_1 A.3.3(24)
UC_O_Acute
in Ada.Characters.Latin_1 A.3.3(24)
UC_O_Circumflex
in Ada.Characters.Latin_1 A.3.3(24)
UC_O_Diaeresis
in Ada.Characters.Latin_1 A.3.3(24)
UC_O_Grave
in Ada.Characters.Latin_1 A.3.3(24)
UC_O_Oblique_Stroke
in Ada.Characters.Latin_1 A.3.3(24)
UC_O_Tilde
in Ada.Characters.Latin_1 A.3.3(24)
UC_U_Acute
in Ada.Characters.Latin_1 A.3.3(24)
UC_U_Circumflex
in Ada.Characters.Latin_1 A.3.3(24)
UC_U_Diaeresis
in Ada.Characters.Latin_1 A.3.3(24)
UC_U_Grave
in Ada.Characters.Latin_1 A.3.3(24)
UC_Y_Acute
in Ada.Characters.Latin_1 A.3.3(24)
UCHAR_MAX
in Interfaces.C B.3(6)
UI 1.3(1.f/5)
ultimate ancestor
of a type 3.4.1(10/2)
unary adding operator 4.5.4(1)
unary operator 4.5(9)
unary_adding_operator 4.5(5)
used 4.4(4), P.1
Unbiased_Rounding attribute A.5.3(39)
Unbounded
child of Ada.Streams.Storage 13.13.1(18/5)
child of Ada.Strings A.4.5(3/5)
child of Ada.Strings.Text_Buffers A.4.12(18/5)
in Ada.Text_IO A.10.1(5)
Unbounded_IO
child of Ada.Text_IO A.10.12(3/5)
Unbounded_Priority_Queues
child of Ada.Containers A.18.30(2/5)
Unbounded_Slice
in Ada.Strings.Unbounded A.4.5(22.1/2), A.4.5(22.2/2)
Unbounded_String
in Ada.Strings.Unbounded A.4.5(4/5)
Unbounded_Synchronized_Queues
child of Ada.Containers A.18.28(2/5)
unchecked storage deallocation 13.11.2(1)
unchecked type conversion 13.9(1)
unchecked union object B.3.3(6/3)
unchecked union subtype B.3.3(6/3)
unchecked union type B.3.3(6/3)
Unchecked_Access attribute 13.10(3), H.4(18)
See also Access attribute 3.10.2(24/1)
Unchecked_Conversion
child of Ada 13.9(3/5)
Unchecked_Deallocation
child of Ada 13.11.2(3/5)
Unchecked_Union aspect B.3.3(3.2/3)
Unchecked_Union pragma J.15.6(2/3), L(37.2/3)
unconditionally evaluated
subexpression 6.1.1(22.11/5)
unconstrained 3.2(9)
object 3.3.1(9/2)
object 6.4.1(16)
subtype 3.2(9), 3.4(6), 3.5(7), 3.5.1(10), 3.5.4(9), 3.5.4(10), 3.5.7(11), 3.5.9(13), 3.5.9(16), 3.6(15/5), 3.6(16), 3.7(26), 3.9(15)
subtype 3.10(14/3)
subtype K.2(33)
unconstrained_array_definition 3.6(3)
used 3.6(2), P.1
undefined result 11.6(5/5)
underline 2.1(15/5)
used 2.4.1(3), 2.4.2(4), P.1
unevaluated
determined to be 6.1.1(24.1/5)
Uniformity Issue (UI) 1.3(1.f/5)
Uniformity Rapporteur Group (URG) 1.3(1.f/5)
Uniformly_Distributed *subtype of* Float
in Ada.Numerics.Float_Random A.5.2(8/5)
uninitialized allocator 4.8(4)
uninitialized variables 13.9.1(2)
[*partial*] 3.3.1(21/5), 13.3(55.i)
union
C B.3.3(1/3)
in Ada.Containers.Hashed_Sets A.18.8(26/5), A.18.8(27/5)
in Ada.Containers.Ordered_Sets A.18.9(27/5), A.18.9(28/5)
unit consistency E.3(6)
unit matrix
complex matrix G.3.2(148/2)
real matrix G.3.1(80/2)
unit vector
complex vector G.3.2(90/2)
real vector G.3.1(48/2)
Unit_Matrix
in Ada.Numerics.Generic_Complex_Arrays G.3.2(51/2)
in Ada.Numerics.Generic_Real_Arrays G.3.1(29/2)
Unit_Vector
in Ada.Numerics.Generic_Complex_Arrays G.3.2(24/2)
in Ada.Numerics.Generic_Real_Arrays G.3.1(14/2)
universal type 3.4.1(6/5)
universal_access
[*partial*] 3.4.1(6/5), 4.2(8/5)
universal_fixed
[*partial*] 3.4.1(6/5), 3.5.6(4)
universal_integer
[*partial*] 3.4.1(6/5), 3.5.4(14), 3.5.4(30), 4.2(8/5)
universal_real
[*partial*] 3.4.1(6/5), 3.5.6(4), 4.2(8/5)
unknown discriminants 3.7(26)
[*partial*] 3.7(1.b/2)
unknown_discriminant_part 3.7(3)
used 3.7(2), P.1
Unknown_Zone_Error
in Ada.Calendar.Time_Zones 9.6.1(5/2)
unmarshalling E.4(9)
unsigned
in Interfaces.C B.3(9)
in Interfaces.COBOL B.4(23)
unsigned type
See modular type 3.5.4(1)
unsigned_char
in Interfaces.C B.3(10)

Unsigned_Conversions
in Ada.Numerics.Big_Numbers.Big_Integers A.5.6(13/5)
unsigned_long
in Interfaces.C B.3(9)
Unsigned_N B.2(8)
unsigned_short
in Interfaces.C B.3(9)
unspecified 1.1.3(18), M.2(1.a)
[*partial*] 2.1(5/5), 3.9(4/2), 3.9(12.5/3), 4.5.2(13), 4.5.2(24.2/1), 4.5.5(21), 4.6(58.5/5), 6.1.1(34/5), 6.1.1(35/3), 6.2(11/3), 7.2(5), 7.6(17.4/3), 9.8(14), 9.10(1/5), 10.2(26), 11.1(6/5), 11.4.1(10.1/4), 11.5(27/2), 13.1(18), 13.7.2(5/2), 13.9.1(7), 13.11(20), 13.11(21.6/5), 13.13.2(36/2), A.1(1/3), A.5.1(34), A.5.2(28), A.5.2(34), A.5.3(41.3/2), A.7(6), A.10(8/5), A.10.7(8/3), A.10.7(12/3), A.10.7(17.3/2), A.10.7(19), A.14(1), A.18(5.v/5), A.18.2(231/3), A.18.2(252/2), A.18.2(83/2), A.18.3(145/3), A.18.3(157/2), A.18.3(55/2), A.18.4(3/2), A.18.4(80/2), A.18.5(43/2), A.18.5(44/2), A.18.5(45/2), A.18.5(46/2), A.18.6(56/3), A.18.6(57/2), A.18.7(3/2), A.18.7(101/2), A.18.7(87/2), A.18.7(88/2), A.18.8(65/2), A.18.8(66.1/3), A.18.8(66/2), A.18.8(67/2), A.18.8(68/2), A.18.8(86/2), A.18.8(87/2), A.18.9(114/2), A.18.9(79.1/3), A.18.9(79/3), A.18.9(80/2), A.18.10(227/3), A.18.10(72/3), A.18.26(5/3), A.18.26(9.4/3), A.18.26(9/3), B.3(46.a.1/1), D.2.2(6.1/2), D.8(19), E.3(5/1), G.1.1(40), G.1.2(33), G.1.2(48), H(4.1), H.2(1/5), K.2(136.4/2)
Unsuppress pragma 11.5(4.1/2), L(37.3/2)
update
the value of an object 3.3(14)
in Interfaces.C.Strings B.3.1(18), B.3.1(19)
Update_Element
in Ada.Containers.Doubly_Linked_Lists A.18.3(17/5)
in Ada.Containers.Hashed_Maps A.18.5(17/5)
in Ada.Containers.Indefinite_Holders A.18.18(15/5)
in Ada.Containers.Multiway_Trees A.18.10(27/5)
in Ada.Containers.Ordered_Maps A.18.6(16/5)
in Ada.Containers.Vectors A.18.2(33/5), A.18.2(34/5)
Update_Element_Preserving_Key
in Ada.Containers.Hashed_Sets A.18.8(58/5)
in Ada.Containers.Ordered_Sets A.18.9(73/5)
Update_Error
in Interfaces.C.Strings B.3.1(20)
upper bound
of a range 3.5(4)
upper-case letter
a category of Character A.3.2(26)
Upper_Case_Map
in Ada.Strings.Maps.Constants A.4.6(5)
Upper_Set
in Ada.Strings.Maps.Constants A.4.6(4)
URG 1.3(1.f/5)
US
in Ada.Characters.Latin_1 A.3.3(6)
usage name 3.1(10)
use clause conflict A(4.k/5)
use-visible 8.3(4), 8.4(9)
use_clause 8.4(2)
used 3.11(4/1), 10.1.2(3), 12.1(5), P.1
Use_Error
in Ada.Direct_IO A.8.4(18)
in Ada.Directories A.16(43/2)
in Ada.IO_Exceptions A.13(4)
in Ada.Sequential_IO A.8.1(15)
in Ada.Streams.Stream_IO A.12.1(26)
in Ada.Text_IO A.10.1(85/5)
Use_Formal aspect H.7.1(3/5)
use_package_clause 8.4(3)
used 8.4(2), P.1
use_type_clause 8.4(4/3)
used 8.4(2), P.1
used
generic formal parameters H.7.1(2/5)
user-defined assignment 7.6(1)
user-defined heap management 13.11(1)
user-defined literal 4.2.1(8/5)
user-defined literal aspect 4.2.1(2/5)
user-defined operator 6.6(1)
user-defined storage management 13.11(1)
UTC_Time_Offset
in Ada.Calendar.Time_Zones 9.6.1(6.1/5)
UTF-16 A.4.11(46/3)
UTF-8 A.4.11(46/3)
UTF_16_Wide_String *subtype of* Wide_String
in Ada.Strings.UTF_Encoding A.4.11(7/3)
UTF_8_String *subtype of* String
in Ada.Strings.UTF_Encoding A.4.11(6/3)
UTF_Encoding
child of Ada.Strings A.4.11(3/5)
UTF_String *subtype of* String
in Ada.Strings.UTF_Encoding A.4.11(5/3)

V

Val attribute 3.5.5(5)
Valid
in Ada.Text_IO.Editing F.3.3(5), F.3.3(12)
in Interfaces.COBOL B.4(33), B.4(38), B.4(43)
Valid attribute 13.9.2(3/4), H(6/5)
Valid_Big_Integer *subtype of* Big_Integer
in Ada.Numerics.Big_Numbers.Big_Integers A.5.6(5/5)
Valid_Big_Real *subtype of* Big_Real
in Ada.Numerics.Big_Numbers.Big_Reals A.5.7(5/5)
value 3.2(10.a)
representation 13.4(10.4/5), K.2(59.4/5)
in Ada.Calendar.Formatting 9.6.1(36/2), 9.6.1(38/2)
in Ada.Environment_Variables A.17(4.1/3), A.17(4/2)
in Ada.Numerics.Discrete_Random A.5.2(26)
in Ada.Numerics.Float_Random A.5.2(14)
in Ada.Strings.Maps A.4.2(21)
in Ada.Strings.Wide_Maps A.4.7(21)
in Ada.Strings.Wide_Wide_Maps A.4.8(21/2)
in Ada.Task_Attributes C.7.2(4)
in Interfaces.C.Pointers B.3.2(6), B.3.2(7)
in Interfaces.C.Strings B.3.1(13), B.3.1(14), B.3.1(15), B.3.1(16)
Value attribute 3.5(52)
value conversion 4.6(5/5)
value_sequence 4.5.10(3/5)
used 4.5.10(2/5), P.1
values
belonging to a subtype 3.2(8/5)
variable 3.3(13/3)
required 5.2(5/2), 6.4.1(5), 12.4(7), 13.11(15/5), 13.11.3(4/3)
variable indexing 4.1.6(16/3)
variable object 3.3(13/3)
variable view 3.3(13/3)
Variable_Indexing aspect 4.1.6(3/5)
variadic
C B.3(60.16/4)
variant 3.8.1(3)
used 3.8.1(2), P.1
See also tagged type 3.9(1)
variant_part 3.8.1(2)
used 3.8(4), P.1
Vector
in Ada.Containers.Vectors A.18.2(8/5), A.18.2(79.2/5)
vector container A.18.2(1/2)
Vector_Iterator_Interfaces
in Ada.Containers.Vectors A.18.2(11.3/5), A.18.2(79.7/5)

Vectors
  child of Ada.Containers A.18.2(6/5)
version
  of a compilation unit E.3(5/1)
Version attribute E.3(3)
vertical line 2.1(15/5)
Vertical_Line
  in Ada.Characters.Latin_1 A.3.3(14)
view 3.1(7)
  of a subtype (implied) 3.1(7.1/3)
  of a type (implied) 3.1(7.1/3)
  of an object (implied) 3.1(7.1/3)
view conversion 4.6(5/5)
view of an entity 1.3.1(86/5)
virtual function
  See dispatching subprogram 3.9.2(1/5)
Virtual_Length
  in Interfaces.C.Pointers B.3.2(13)
visibility
  direct 8.3(2), 8.3(21)
  immediate 8.3(4), 8.3(21)
  use clause 8.3(4), 8.4(9)
visibility rules 8.3(1)
visible 8.3(2), 8.3(14)
  aspect_specification 8.3(23.1/3)
  attribute_definition_clause 8.3(23.1/3)
  within a pragma in a context_clause 10.1.6(3)
  within a pragma that appears at the place of a compilation unit 10.1.6(5)
  within a use_clause in a context_clause 10.1.6(3)
  within a with_clause 10.1.6(2/2)
  within the parent_unit_name of a library unit 10.1.6(2/2)
  within the parent_unit_name of a subunit 10.1.6(4)
visible part 8.2(5)
  of a formal package 12.7(10/2)
  of a generic unit 8.2(8)
  of a package (other than a generic formal package) 7.1(6/2)
  of a protected unit 9.4(11/2)
  of a task unit 9.1(9)
  of a view of a callable entity 8.2(6)
  of a view of a composite type 8.2(7)
  of an instance 12.3(12.b)
visibly of an access type H.4(23.6/5)
volatile C.6(8/3)
Volatile aspect C.6(6.4/3)
Volatile pragma J.15.8(3/3), L(38.1/3)
Volatile_Components aspect C.6(6.8/3)
Volatile_Components pragma J.15.8(6/3), L(39.1/3)
VT
  in Ada.Characters.Latin_1 A.3.3(5)
VTS
  in Ada.Characters.Latin_1 A.3.3(17)

W

Wait_For_Release
  in Ada.Synchronous_Barriers D.10.1(6/5)
wchar_array
  in Interfaces.C B.3(33/3)
wchar_t
  in Interfaces.C B.3(30/1)
Wednesday
  in Ada.Calendar.Formatting 9.6.1(17/2)
well-formed picture String
  for edited output F.3.1(1/3)
Wide_Bounded
  child of Ada.Strings A.4.7(1/3)
Wide_Bounded_IO
  child of Ada.Wide_Text_IO A.11(4/3)
Wide_Character 3.5.2(3/5)
  in Standard A.1(36.1/5)
Wide_Character_Mapping
  in Ada.Strings.Wide_Maps A.4.7(20/5)
Wide_Character_Mapping_Function
  in Ada.Strings.Wide_Maps A.4.7(26)
Wide_Character_Range
  in Ada.Strings.Wide_Maps A.4.7(6)
Wide_Character_Ranges
  in Ada.Strings.Wide_Maps A.4.7(7)
Wide_Character_Sequence *subtype of* Wide_String
  in Ada.Strings.Wide_Maps A.4.7(16)
Wide_Character_Set
  in Ada.Strings.Wide_Maps A.4.7(4/5)
  in Ada.Strings.Wide_Maps.Wide_-Constants A.4.8(48/2)
Wide_Characters
  child of Ada A.3.1(4/5)
Wide_Command_Line
  child of Ada A.15.1(2/5)
Wide_Constants
  child of Ada.Strings.Wide_Maps A.4.7(1/3), A.4.8(28/2)
Wide_Directories
  child of Ada A.16.2(2/5)
Wide_Environment_Variables
  child of Ada A.17.1(2/5)
Wide_Equal_Case_Insensitive
  child of Ada.Strings A.4.7(1/3)
  child of Ada.Strings.Wide_Bounded A.4.7(1/3)
  child of Ada.Strings.Wide_Fixed A.4.7(1/3)
  child of Ada.Strings.Wide_Unbounded A.4.7(1/3)
Wide_Exception_Name
  in Ada.Exceptions 11.4.1(2/5), 11.4.1(5/2)
Wide_Expanded_Name
  in Ada.Tags 3.9(7/2)
Wide_File_Names
  in Ada.Direct_IO A.8.4(18.1/5)
  in Ada.Sequential_IO A.8.1(15.1/5)
  in Ada.Streams.Stream_IO A.12.1(26.1/5)
  in Ada.Text_IO A.10.1(85.1/5)
Wide_Fixed
  child of Ada.Strings A.4.7(1/3)
Wide_Get
  in Ada.Strings.Text_Buffers.Unbounded A.4.12(21/5)
Wide_Get_UTF_16
  in Ada.Strings.Text_Buffers.Unbounded A.4.12(24/5)
Wide_Hash
  child of Ada.Strings A.4.7(1/3)
  child of Ada.Strings.Wide_Bounded A.4.7(1/3)
  child of Ada.Strings.Wide_Fixed A.4.7(1/3)
  child of Ada.Strings.Wide_Unbounded A.4.7(1/3)
Wide_Hash_Case_Insensitive
  child of Ada.Strings A.4.7(1/3)
  child of Ada.Strings.Wide_Bounded A.4.7(1/3)
  child of Ada.Strings.Wide_Fixed A.4.7(1/3)
  child of Ada.Strings.Wide_Unbounded A.4.7(1/3)
Wide_Image attribute 4.10(31/5), 4.10(39/5)
Wide_Maps
  child of Ada.Strings A.4.7(3/5)
wide_nul
  in Interfaces.C B.3(31/1)
Wide_Put
  in Ada.Strings.Text_Buffers A.4.12(8/5)
Wide_Put_UTF_16
  in Ada.Strings.Text_Buffers A.4.12(11/5)
Wide_Space
  in Ada.Strings A.4.1(4/2)
Wide_String
  in Standard A.1(41/3)
Wide_Strings
  child of Ada.Strings.UTF_Encoding A.4.11(30/5)
Wide_Text_IO
  child of Ada A.11(2/2)
Wide_Unbounded
  child of Ada.Strings A.4.7(1/3)
Wide_Unbounded_IO
  child of Ada.Wide_Text_IO A.11(5/3)
Wide_Value attribute 3.5(40)
Wide_Wide_Bounded
  child of Ada.Strings A.4.8(1/3)
Wide_Wide_Bounded_IO
  child of Ada.Wide_Wide_Text_IO A.11(4/3)
Wide_Wide_Character 3.5.2(4/5)
  in Standard A.1(36.2/5)
Wide_Wide_Character_Mapping
  in Ada.Strings.Wide_Wide_Maps A.4.8(20/5)
Wide_Wide_Character_Mapping_Function
  in Ada.Strings.Wide_Wide_Maps A.4.8(26/2)

Wide_Wide_Character_Range
  in Ada.Strings.Wide_Wide_Maps A.4.8(6/2)
Wide_Wide_Character_Ranges
  in Ada.Strings.Wide_Wide_Maps A.4.8(7/2)
Wide_Wide_Character_Sequence *subtype of* Wide_Wide_String
  in Ada.Strings.Wide_Wide_Maps A.4.8(16/2)
Wide_Wide_Character_Set
  in Ada.Strings.Wide_Wide_Maps A.4.8(4/5)
Wide_Wide_Characters
  child of Ada A.3.1(6/5)
Wide_Wide_Command_Line
  child of Ada A.15.1(3/5)
Wide_Wide_Constants
  child of Ada.Strings.Wide_Wide_Maps A.4.8(1/3)
Wide_Wide_Directories
  child of Ada A.16.2(3/5)
Wide_Wide_Environment_Variables
  child of Ada A.17.1(3/5)
Wide_Wide_Equal_Case_Insensitive
  child of Ada.Strings A.4.8(1/3)
  child of Ada.Strings.Wide_Wide_-Bounded A.4.8(1/3)
  child of Ada.Strings.Wide_Wide_Fixed A.4.8(1/3)
  child of Ada.Strings.Wide_Wide_-Unbounded A.4.8(1/3)
Wide_Wide_Exception_Name
  in Ada.Exceptions 11.4.1(2/5), 11.4.1(5/2)
Wide_Wide_Expanded_Name
  in Ada.Tags 3.9(7/2)
Wide_Wide_File_Names
  in Ada.Direct_IO A.8.4(18.8/5)
  in Ada.Sequential_IO A.8.1(15.8/5)
  in Ada.Streams.Stream_IO A.12.1(26.8/5)
  in Ada.Text_IO A.10.1(85.8/5)
Wide_Wide_Fixed
  child of Ada.Strings A.4.8(1/3)
Wide_Wide_Get
  in Ada.Strings.Text_Buffers.Unbounded A.4.12(22/5)
Wide_Wide_Hash
  child of Ada.Strings A.4.8(1/3)
  child of Ada.Strings.Wide_Wide_-Bounded A.4.8(1/3)
  child of Ada.Strings.Wide_Wide_Fixed A.4.8(1/3)
  child of Ada.Strings.Wide_Wide_-Unbounded A.4.8(1/3)
Wide_Wide_Hash_Case_Insensitive
  child of Ada.Strings A.4.8(1/3)
  child of Ada.Strings.Wide_Wide_-Bounded A.4.8(1/3)
  child of Ada.Strings.Wide_Wide_Fixed A.4.8(1/3)
  child of Ada.Strings.Wide_Wide_-Unbounded A.4.8(1/3)
Wide_Wide_Image attribute 4.10(30/5), 4.10(38/5)
Wide_Wide_Maps
  child of Ada.Strings A.4.8(3/5)
Wide_Wide_Put
  in Ada.Strings.Text_Buffers A.4.12(9/5)
Wide_Wide_Space
  in Ada.Strings A.4.1(4/2)
Wide_Wide_String
  in Standard A.1(42.1/3)
Wide_Wide_Strings
  child of Ada.Strings.UTF_Encoding A.4.11(38/5)
Wide_Wide_Text_IO
  child of Ada A.11(3/2)
Wide_Wide_Unbounded
  child of Ada.Strings A.4.8(1/3)
Wide_Wide_Unbounded_IO
  child of Ada.Wide_Wide_Text_IO A.11(5/3)
Wide_Wide_Value attribute 3.5(39.1/2)
Wide_Wide_Width attribute 3.5(37.1/5)
Wide_Width attribute 3.5(38/5)
Width attribute 3.5(39/5)
with_clause 10.1.2(4/2)
  mentioned in 10.1.2(6/2)
  named in 10.1.2(6/2)
  used 10.1.2(3), P.1
within
  immediately 8.1(13)
word 13.3(8)
Word_Size
  in System 13.7(13)
wording changes from Ada 2005 1.1.2(39.jj/3)
wording changes from Ada 2012 1.1.2(39.ww/5)
wording changes from Ada 83 1.1.2(39.j/2)
wording changes from Ada 95 1.1.2(39.w/2)
Write
  in Ada.Direct_IO A.8.4(13/5)
  in Ada.Sequential_IO A.8.1(12/5)
  in Ada.Storage_IO A.9(7)
  in Ada.Streams 13.13.1(6)
  in Ada.Streams.Storage.Bounded 13.13.1(29/5)
  in Ada.Streams.Storage.Unbounded 13.13.1(21/5)
  in Ada.Streams.Stream_IO A.12.1(18/5), A.12.1(19/5)
  in System.RPC E.5(8)
Write aspect 13.13.2(38/5)
Write attribute 13.13.2(3), 13.13.2(11)
Write clause 13.3(7/2), 13.13.2(38/5)
Write'Class aspect 13.13.2(38/5)

X

xor operator 4.5(1), 4.5.1(2)

Y

Year
  in Ada.Calendar 9.6(13)
  in Ada.Calendar.Formatting 9.6.1(21/2)
Year_Number *subtype of* Integer
  in Ada.Calendar 9.6(11/2)
Yen_Sign
  in Ada.Characters.Latin_1 A.3.3(21/3)
Yield
  in Ada.Dispatching D.2.1(1.3/5)
Yield aspect D.2.1(1.7/5)
Yield_To_Higher
  in Ada.Dispatching.Non_Preemptive D.2.4(2.2/5)
Yield_To_Same_Or_Higher
  in Ada.Dispatching.Non_Preemptive D.2.4(2.2/5)