INTERNATIONAL STANDARD

ISO/IEC 8652

Second edition
1995-02-15

withdrawn

Information technology — Programming languages — Ada

Technologies de l'information — Langages de programmation — Ada

Reference number
ISO/IEC 8652:1995(E)

Contents

Foreword x
Introduction xi
1. General 1
1.1 Scope 1
1.1.1 Extent 1
1.1.2 Structure 2
1.1.3 Conformity of an Implementation with the Standard 4
1.1.4 Method of Description and Syntax Notation 5
1.1.5 Classification of Errors 7
1.2 Normative References 8
1.3 Definitions 8
2. Lexical Elements 9
2.1 Character Set 9
2.2 Lexical Elements, Separators, and Delimiters 10
2.3 Identifiers 11
2.4 Numeric Literals 12
2.4.1 Decimal Literals 12
2.4.2 Based Literals 12
2.5 Character Literals 13
2.6 String Literals 13
2.7 Comments 14
2.8 Pragmas 14
2.9 Reserved Words 17
3. Declarations and Types 19
3.1 Declarations 19
3.2 Types and Subtypes 20
3.2.1 Type Declarations 21
3.2.2 Subtype Declarations 23
3.2.3 Classification of Operations 24
3.3 Objects and Named Numbers 24
3.3.1 Object Declarations 26
3.3.2 Number Declarations 28
3.4 Derived Types and Classes 28
3.4.1 Derivation Classes 31

Withdrawn

© ISO/IEC 1995
All rights reserved. Unless otherwise specified, no part of this publication may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying and microfilm, without permission in writing from the publisher.

ISO/IEC Copyright Office • Case Postale 56 • CH-1211 Genève 20 • Switzerland

Printed in Switzerland

3.5 Scalar Types ... 32
3.5.1 Enumeration Types ... 36
3.5.2 Character Types ... 37
3.5.3 Boolean Types ... 38
3.5.4 Integer Types ... 38
3.5.5 Operations of Discrete Types ... 41
3.5.6 Real Types ... 42
3.5.7 Floating Point Types ... 43
3.5.8 Operations of Floating Point Types ... 44
3.5.9 Fixed Point Types ... 45
3.5.10 Operations of Fixed Point Types ... 47
3.6 Array Types ... 48
3.6.1 Index Constraints and Discrete Ranges ... 50
3.6.2 Operations of Array Types ... 51
3.6.3 String Types ... 52
3.7 Discriminants ... 52
3.7.1 Discriminant Constraints ... 55
3.7.2 Operations of Discriminated Types ... 56
3.8 Record Types ... 56
3.8.1 Variant Parts and Discrete Choices ... 59
3.9 Tagged Types and Type Extensions ... 60
3.9.1 Type Extensions ... 62
3.9.2 Dispatching Operations of Tagged Types ... 63
3.9.3 Abstract Types and Subprograms ... 65
3.10 Access Types ... 67
3.10.1 Incomplete Type Declarations ... 69
3.10.2 Operations of Access Types ... 70
3.11 Declarative Parts ... 73
3.11.1 Completions of Declarations ... 74
4. Names and Expressions ... 75
4.1 Names ... 75
4.1.1 Indexed Components ... 76
4.1.2 Slices ... 77
4.1.3 Selected Components ... 78
4.1.4 Attributes ... 79
4.2 Literals ... 80
4.3 Aggregates ... 81
4.3.1 Record Aggregates ... 82
4.3.2 Extension Aggregates ... 83
4.3.3 Array Aggregates ... 84
4.4 Expressions ... 87
4.5 Operators and Expression Evaluation ... 88
4.5.1 Logical Operators and Short-circuit Control Forms ... 89
4.5.2 Relational Operators and Membership Tests ... 90
4.5.3 Binary Adding Operators ... 93
4.5.4 Unary Adding Operators ... 94
4.5.5 Multiplying Operators ... 94
4.5.6 Highest Precedence Operators ... 96
4.6 Type Conversions ... 97
4.7 Qualified Expressions ... 101
4.8 Allocators ... 102
4.9 Static Expressions and Static Subtypes ... 103
4.9.1 Statically Matching Constraints and Subtypes ... 105

Withdrawn

5. Statements 107
5.1 Simple and Compound Statements - Sequences of Statements 107
5.2 Assignment Statements 108
5.3 If Statements 110
5.4 Case Statements 110
5.5 Loop Statements 112
5.6 Block Statements 113
5.7 Exit Statements 114
5.8 Goto Statements 115
6. Subprograms 117
6.1 Subprogram Declarations 117
6.2 Formal Parameter Modes 119
6.3 Subprogram Bodies 120
6.3.1 Conformance Rules 121
6.3.2 Inline Expansion of Subprograms 122
6.4 Subprogram Calls 123
6.4.1 Parameter Associations 124
6.5 Return Statements 125
6.6 Overloading of Operators 127
7. Packages 129
7.1 Package Specifications and Declarations 129
7.2 Package Bodies 130
7.3 Private Types and Private Extensions 131
7.3.1 Private Operations 133
7.4 Deferred Constants 135
7.5 Limited Types 136
7.6 User-Defined Assignment and Finalization 137
7.6.1 Completion and Finalization 139
8. Visibility Rules 143
8.1 Declarative Region 143
8.2 Scope of Declarations 144
8.3 Visibility 145
8.4 Use Clauses 147
8.5 Renaming Declarations 148
8.5.1 Object Renaming Declarations 148
8.5.2 Exception Renaming Declarations 149
8.5.3 Package Renaming Declarations 149
8.5.4 Subprogram Renaming Declarations 150
8.5.5 Generic Renaming Declarations 151
8.6 The Context of Overload Resolution 151
9. Tasks and Synchronization 155
9.1 Task Units and Task Objects 155
9.2 Task Execution - Task Activation 157
9.3 Task Dependence - Termination of Tasks 158
9.4 Protected Units and Protected Objects 159
9.5 Intertask Communication 162
9.5.1 Protected Subprograms and Protected Actions 163
9.5.2 Entries and Accept Statements 164
9.5.3 Entry Calls 167
9.5.4 Requeue Statements 169
9.6 Delay Statements, Duration, and Time 171
9.7 Select Statements 173

Withdrawn

9.7.1 Selective Accept ... 174
9.7.2 Timed Entry Calls ... 176
9.7.3 Conditional Entry Calls ... 176
9.7.4 Asynchronous Transfer of Control ... 177
9.8 Abort of a Task - Abort of a Sequence of Statements ... 178
9.9 Task and Entry Attributes ... 179
9.10 Shared Variables ... 180
9.11 Example of Tasking and Synchronization ... 181
10. Program Structure and Compilation Issues ... 183
10.1 Separate Compilation ... 183
10.1.1 Compilation Units - Library Units ... 183
10.1.2 Context Clauses - With Clauses ... 186
10.1.3 Subunits of Compilation Units ... 186
10.1.4 The Compilation Process ... 188
10.1.5 Pragmas and Program Units ... 189
10.1.6 Environment-Level Visibility Rules ... 190
10.2 Program Execution ... 191
10.2.1 Elaboration Control ... 193
11. Exceptions ... 195
11.1 Exception Declarations ... 195
11.2 Exception Handlers ... 195
11.3 Raise Statements ... 196
11.4 Exception Handling ... 197
11.4.1 The Package Exceptions ... 197
11.4.2 Example of Exception Handling ... 199
11.5 Suppressing Checks ... 200
11.6 Exceptions and Optimization ... 202
12. Generic Units ... 205
12.1 Generic Declarations ... 205
12.2 Generic Bodies ... 206
12.3 Generic Instantiation ... 207
12.4 Formal Objects ... 210
12.5 Formal Types ... 211
12.5.1 Formal Private and Derived Types ... 212
12.5.2 Formal Scalar Types ... 213
12.5.3 Formal Array Types ... 214
12.5.4 Formal Access Types ... 215
12.6 Formal Subprograms ... 215
12.7 Formal Packages ... 217
12.8 Example of a Generic Package ... 217
13. Representation Issues ... 219
13.1 Representation Items ... 219
13.2 Pragma Pack ... 221
13.3 Representation Attributes ... 222
13.4 Enumeration Representation Clauses ... 227
13.5 Record Layout ... 228
13.5.1 Record Representation Clauses ... 228
13.5.2 Storage Place Attributes ... 230
13.5.3 Bit Ordering ... 230
13.6 Change of Representation ... 231
13.7 The Package System ... 232
13.7.1 The Package System.Storage_Elements ... 234

Withdrawn

13.7.2 The Package System.Address_To_Access_Conversions . . . 234
13.8 Machine Code Insertions . . . 235
13.9 Unchecked Type Conversions . . . 236
13.9.1 Data Validity . . . 237
13.9.2 The Valid Attribute . . . 238
13.10 Unchecked Access Value Creation . . . 238
13.11 Storage Management . . . 239
13.11.1 The Max_Size_In_Storage_Elements Attribute . . . 242
13.11.2 Unchecked Storage Deallocation . . . 242
13.11.3 Pragma Controlled . . . 243
13.12 Pragma Restrictions . . . 243
13.13 Streams . . . 244
13.13.1 The Package Streams . . . 244
13.13.2 Stream-Oriented Attributes . . . 245
13.14 Freezing Rules . . . 247

ANNEXES

A. Predefined Language Environment . . . 251
A.1 The Package Standard . . . 252
A.2 The Package Ada . . . 255
A.3 Character Handling . . . 255
A.3.1 The Package Characters . . . 256
A.3.2 The Package Characters.Handling . . . 256
A.3.3 The Package Characters.Latin_1 . . . 258
A.4 String Handling . . . 262
A.4.1 The Package Strings . . . 263
A.4.2 The Package Strings.Maps . . . 263
A.4.3 Fixed-Length String Handling . . . 266
A.4.4 Bounded-Length String Handling . . . 273
A.4.5 Unbounded-Length String Handling . . . 278
A.4.6 String-Handling Sets and Mappings . . . 283
A.4.7 Wide_String Handling . . . 283
A.5 The Numerics Packages . . . 285
A.5.1 Elementary Functions . . . 286
A.5.2 Random Number Generation . . . 289
A.5.3 Attributes of Floating Point Types . . . 293
A.5.4 Attributes of Fixed Point Types . . . 297
A.6 Input-Output . . . 297
A.7 External Files and File Objects . . . 298
A.8 Sequential and Direct Files . . . 299
A.8.1 The Generic Package Sequential_IO . . . 299
A.8.2 File Management . . . 300
A.8.3 Sequential Input-Output Operations . . . 302
A.8.4 The Generic Package Direct_IO . . . 303
A.8.5 Direct Input-Output Operations . . . 304
A.9 The Generic Package Storage_IO . . . 305
A.10 Text Input-Output . . . 305
A.10.1 The Package Text_IO . . . 307
A.10.2 Text File Management . . . 311
A.10.3 Default Input, Output, and Error Files . . . 312
A.10.4 Specification of Line and Page Lengths . . . 313

Withdrawn

A.10.5 Operations on Columns, Lines, and Pages . . . 314
A.10.6 Get and Put Procedures . . . 317
A.10.7 Input-Output of Characters and Strings . . . 318
A.10.8 Input-Output for Integer Types . . . 320
A.10.9 Input-Output for Real Types . . . 322
A.10.10 Input-Output for Enumeration Types . . . 325
A.11 Wide Text Input-Output . . . 326
A.12 Stream Input-Output . . . 326
A.12.1 The Package Streams.Stream_IO . . . 326
A.12.2 The Package Text_IO.Text_Streams . . . 328
A.12.3 The Package Wide_Text_IO.Text_Streams . . . 329
A.13 Exceptions in Input-Output . . . 329
A.14 File Sharing . . . 330
A.15 The Package Command_Line . . . 331

B. Interface to Other Languages . . . 333
B.1 Interfacing Pragmas . . . 333
B.2 The Package Interfaces . . . 336
B.3 Interfacing with C . . . 337
B.3.1 The Package Interfaces.C.Strings . . . 341
B.3.2 The Generic Package Interfaces.C.Pointers . . . 344
B.4 Interfacing with COBOL . . . 347
B.5 Interfacing with Fortran . . . 353

C. Systems Programming . . . 357
C.1 Access to Machine Operations . . . 357
C.2 Required Representation Support . . . 358
C.3 Interrupt Support . . . 358
C.3.1 Protected Procedure Handlers . . . 360
C.3.2 The Package Interrupts . . . 362
C.4 Preelaboration Requirements . . . 364
C.5 Pragma Discard_Names . . . 365
C.6 Shared Variable Control . . . 365
C.7 Task Identification and Attributes . . . 367
C.7.1 The Package Task_Identification . . . 367
C.7.2 The Package Task_Attributes . . . 368

D. Real-Time Systems . . . 371
D.1 Task Priorities . . . 371
D.2 Priority Scheduling . . . 373
D.2.1 The Task Dispatching Model . . . 373
D.2.2 The Standard Task Dispatching Policy . . . 375
D.3 Priority Ceiling Locking . . . 376
D.4 Entry Queuing Policies . . . 378
D.5 Dynamic Priorities . . . 379
D.6 Preemptive Abort . . . 380
D.7 Tasking Restrictions . . . 381
D.8 Monotonic Time . . . 382
D.9 Delay Accuracy . . . 386
D.10 Synchronous Task Control . . . 387
D.11 Asynchronous Task Control . . . 387
D.12 Other Optimizations and Determinism Rules . . . 388

E. Distributed Systems . . . 391
E.1 Partitions . . . 391
E.2 Categorization of Library Units . . . 393

Withdrawn

E.2.1 Shared Passive Library Units . . . 394
E.2.2 Remote Types Library Units . . . 394
E.2.3 Remote Call Interface Library Units . . . 395
E.3 Consistency of a Distributed System . . . 396
E.4 Remote Subprogram Calls . . . 397
E.4.1 Pragma Asynchronous . . . 398
E.4.2 Example of Use of a Remote Access-to-Class-Wide Type . . . 399
E.5 Partition Communication Subsystem . . . 401
F. Information Systems . . . 403
F.1 Machine_Radix Attribute Definition Clause . . . 403
F.2 The Package Decimal . . . 404
F.3 Edited Output for Decimal Types . . . 405
F.3.1 Picture String Formation . . . 406
F.3.2 Edited Output Generation . . . 409
F.3.3 The Package Text_IO.Editing . . . 414
F.3.4 The Package Wide_Text_IO.Editing . . . 417
G. Numerics . . . 419
G.1 Complex Arithmetic . . . 419
G.1.1 Complex Types . . . 419
G.1.2 Complex Elementary Functions . . . 424
G.1.3 Complex Input-Output . . . 427
G.1.4 The Package Wide_Text_IO.Complex_IO . . . 429
G.2 Numeric Performance Requirements . . . 430
G.2.1 Model of Floating Point Arithmetic . . . 430
G.2.2 Model-Oriented Attributes of Floating Point Types . . . 431
G.2.3 Model of Fixed Point Arithmetic . . . 432
G.2.4 Accuracy Requirements for the Elementary Functions . . . 434
G.2.5 Performance Requirements for Random Number Generation . . . 436
G.2.6 Accuracy Requirements for Complex Arithmetic . . . 436
H. Safety and Security . . . 439
H.1 Pragma Normalize_Scalars . . . 439
H.2 Documentation of Implementation Decisions . . . 440
H.3 Reviewable Object Code . . . 440
H.3.1 Pragma Reviewable . . . 440
H.3.2 Pragma Inspection_Point . . . 441
H.4 Safety and Security Restrictions . . . 442
J. Obsolescent Features . . . 445
J.1 Renamings of Ada 83 Library Units . . . 445
J.2 Allowed Replacements of Characters . . . 445
J.3 Reduced Accuracy Subtypes . . . 446
J.4 The Constrained Attribute . . . 446
J.5 ASCII . . . 447
J.6 Numeric_Error . . . 447
J.7 At Clauses . . . 447
J.7.1 Interrupt Entries . . . 448
J.8 Mod Clauses . . . 449
J.9 The Storage_Size Attribute . . . 449
K. Language-Defined Attributes . . . 451
L. Language-Defined Pragmas . . . 465

Withdrawn

M. Implementation-Defined Characteristics 467
N. Glossary 473
P. Syntax Summary 477
Index 501

withdrawn

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work.

In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75 % of the national bodies casting a vote.

International Standard ISO/IEC 8652 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information Technology*.

This second edition cancels and replaces the first edition (ISO 8652:1987), of which it constitutes a technical revision.

Annexes A to J form an integral part of this International Standard. Annexes K to P are for information only.

Withdrawn

Introduction

This is the Ada Reference Manual.

Other available Ada documents include:

- Rationale for the Ada Programming Language — 1995 edition, which gives an introduction to the new features of Ada, and explains the rationale behind them. Programmers should read this first.
- Changes to Ada — 1987 to 1995. This document lists in detail the changes made to the 1987 edition of the standard.
- The Annotated Ada Reference Manual (AARM). The AARM contains all of the text in the RM95, plus various annotations. It is intended primarily for compiler writers, validation test writers, and others who wish to study the fine details. The annotations include detailed rationale for individual rules and explanations of some of the more arcane interactions among the rules.

Design Goals

Ada was originally designed with three overriding concerns: program reliability and maintenance, programming as a human activity, and efficiency. This 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.

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.

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.

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.

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 implementation

techniques. Any proposed construct whose implementation was unclear or that required excessive machine resources was rejected.

Language Summary

An Ada program is composed of one or more program units. Program units may 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 must be visible to other units, and a body, containing the implementation details, which need not be visible to other units. Most program units can be compiled separately.

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.

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 must name the library units it requires.

Program Units

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 may read data, update variables, or produce some output. It may 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.

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.

Subprogram and package units may 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.

A task unit is the basic unit for defining a task whose sequence of actions may be executed concurrently with those of other tasks. Such tasks may be implemented on multicomputers, multiprocessors, or with interleaved execution on a single processor. A task unit may define either a single executing task or a task type permitting the creation of any number of similar tasks.

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 must be true before the body of the entry is executed. A protected unit may define a single protected object or a protected type permitting the creation of several similar objects.

Declarations and Statements

Withdrawn

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.

The declarative part associates names with declared entities. For example, a name may 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.

The sequence of statements describes a sequence of actions that are to be performed. The statements are executed in succession (unless a transfer of control causes execution to continue from another place).

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.

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.

The loop statement provides the basic iterative mechanism in the language. A loop statement specifies that a sequence of statements is to be executed repeatedly as directed by an iteration scheme, or until an exit statement is encountered.

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

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 may 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.

Execution of a program unit may encounter error situations in which normal program execution cannot continue. For example, an arithmetic computation may exceed the maximum allowed value of a number, or an attempt may 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.

Data Types

Every object in the language has a type, which characterizes a set of values and a set of applicable operations. The main classes of types are elementary types (comprising enumeration, numeric, and access types) and composite types (including array and record types).

Withdrawn

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, and Wide_Character are predefined.

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.

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 and Wide_String are predefined.

Record, task, and protected types may 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.

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 may designate the same object, and components of one object may 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.

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.

From any type a new type may be defined by derivation. A type, together with its derivatives (both direct and indirect) form a derivation class. Class-wide operations may be defined that accept as a parameter an operand of any type in a derivation class. For record and private types, the derivatives may be extensions of the parent type. Types that support these object-oriented capabilities of class-wide operations and type extension must be 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.

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.

Other Facilities

Representation 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 must 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.

The predefined environment of the language provides for input-output and other capabilities (such as string manipulation and random number generation) 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. Other standard library packages are defined in annexes of the standard to support systems with specialized requirements.

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.

Language Changes

This International Standard replaces the first edition of 1987. In this edition, the following major language changes have been incorporated:

- Support for standard 8-bit and 16-bit character sets. See Section 2, 3.5.2, 3.6.3, A.1, A.3, and A.4.
- Object-oriented programming with run-time polymorphism. See the discussions of classes, derived types, tagged types, record extensions, and private extensions in clauses 3.4, 3.9, and 7.3. See also the new forms of generic formal parameters that are allowed by 12.5.1, ''Formal Private and Derived Types'' and 12.7, ''Formal Packages''.
- Access types have been extended to allow an access value to designate a subprogram or an object declared by an object declaration (as opposed to just a heap-allocated object). See 3.10.
- Efficient data-oriented synchronization is provided via protected types. See Section 9.
- The library units of a library may be organized into a hierarchy of parent and child units. See Section 10.
- Additional support has been added for interfacing to other languages. See Annex B.
- The Specialized Needs Annexes have been added to provide specific support for certain application areas:
  - Annex C, ''Systems Programming''
  - Annex D, ''Real-Time Systems''
  - Annex E, ''Distributed Systems''
  - Annex F, ''Information Systems''
  - Annex G, ''Numerics''
  - Annex H, ''Safety and Security''

Withdrawn

Instructions for Comment Submission

Informal comments on this International Standard may be sent via e-mail to **ada-comment@sei.cmu.edu.** If appropriate, the Project Editor will initiate the defect correction procedure.

Comments should use the following format:

> **!topic** *Title summarizing comment*
> **!reference** RM95-*ss.ss(pp)*
> **!from** *Author Name yy-mm-dd*
> **!keywords** *keywords related to topic*
> **!discussion**
>
> *text of discussion*

where *ss.ss* is the section, 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.

Multiple comments per e-mail message are acceptable. Please use a descriptive "Subject" in your e-mail message.

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:

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

Formal requests for interpretations and for reporting defects in this International Standard may be made in accordance with the ISO/IEC JTC 1 Directives and the ISO/IEC JTC 1/SC 22 policy for interpretations. National Bodies may submit a Defect Report to ISO/IEC JTC 1/SC 22 for resolution under the JTC 1 procedures. A response will be provided and, if appropriate, a Technical Corrigendum will be issued in accordance with the procedures.

Withdrawn

Information technology — Programming languages — Ada

Withdrawn

Section 1: General

Ada is a programming language designed to support the construction of long-lived, highly reliable software systems. The language includes facilities to define packages of related types, objects, and operations. The packages may be parameterized and the types may be extended to support the construction of libraries of reusable, adaptable software components. The operations may be implemented as subprograms using conventional sequential control structures, or as entries that include synchronization of concurrent threads of control as part of their invocation. The language treats modularity in the physical sense as well, with a facility to support separate compilation.

The language includes a complete facility for the support of real-time, concurrent programming. Errors can be signaled as exceptions and handled explicitly. The language also covers systems programming; this requires precise control over the representation of data and access to system-dependent properties. Finally, a predefined environment of standard packages is provided, including facilities for, among others, input-output, string manipulation, numeric elementary functions, and random number generation.

1.1 Scope

This International Standard specifies the form and meaning of programs written in Ada. Its purpose is to promote the portability of Ada programs to a variety of data processing systems.

1.1.1 Extent

This International Standard specifies:

- The form of a program written in Ada;
- The effect of translating and executing such a program;

- The manner in which program units may be combined to form Ada programs;
- The language-defined library units that a conforming implementation is required to supply;
- The permissible variations within the standard, and the manner in which they are to be documented;
- Those violations of the standard that a conforming implementation is required to detect, and the effect of attempting to translate or execute a program containing such violations;
- Those violations of the standard that a conforming implementation is not required to detect.

This International Standard does not specify:

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

1.1.2 Structure

This International Standard contains thirteen sections, fourteen annexes, and an index.

The *core* of the Ada language consists of:

- Sections 1 through 13
- Annex A, ''Predefined Language Environment''
- Annex B, ''Interface to Other Languages''
- Annex J, ''Obsolescent Features''

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

- Annex C, ''Systems Programming''
- Annex D, ''Real-Time Systems''
- Annex E, ''Distributed Systems''
- Annex F, ''Information Systems''
- Annex G, ''Numerics''
- Annex H, ''Safety and Security''

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

- Text under a NOTES or Examples heading.

Withdrawn

- Each clause or subclause whose title starts with the word ''Example'' or ''Examples''.

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

The following Annexes are informative:

- Annex K, ''Language-Defined Attributes''
- Annex L, ''Language-Defined Pragmas''
- Annex M, ''Implementation-Defined Characteristics''
- Annex N, ''Glossary''
- Annex P, ''Syntax Summary''

Each section is divided into clauses and subclauses that have a common structure. Each section, clause, and subclause first introduces its subject. After the introductory text, text is labeled with the following headings:

Syntax

Syntax rules (indented).

Name Resolution Rules

Compile-time rules that are used in name resolution, including overload resolution.

Legality Rules

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

Static Semantics

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

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.

Dynamic Semantics

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

Bounded (Run-Time) Errors

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

Erroneous Execution

Situations that result in erroneous execution (see 1.1.5).

Implementation Requirements

Additional requirements for conforming implementations.

Documentation Requirements

Documentation requirements for conforming implementations.

Metrics

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

Withdrawn

Implementation Permissions

Additional permissions given to the implementer.

Implementation Advice

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.

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

Examples

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

1.1.3 Conformity of an Implementation with the Standard

Implementation Requirements

A conforming implementation shall:

- Translate and correctly execute legal programs written in Ada, provided that they are not so large as to exceed the capacity of the implementation;
- 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);
- Identify all programs or program units that contain errors whose detection is required by this International Standard;
- Supply all language-defined library units required by this International Standard;
- Contain no variations except those explicitly permitted by this International Standard, or those that are impossible or impractical to avoid given the implementation's execution environment;
- Specify all such variations in the manner prescribed by this International Standard.

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*:

- Any interaction with an external file (see A.7);
- The execution of certain code_statements (see 13.8); which code_statements cause external interactions is implementation defined.
- Any call on an imported subprogram (see Annex B), including any parameters passed to it;
- Any result returned or exception propagated from a main subprogram (see 10.2) or an exported subprogram (see Annex B) to an external caller;
- Any read or update of an atomic or volatile object (see C.6);
- The values of imported and exported objects (see Annex B) at the time of any other interaction with the external environment.

A conforming implementation of this International Standard 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 International Standard for the semantics of the given program.

Withdrawn

An implementation that conforms to this Standard shall support each capability required by the core language as specified. In addition, an implementation that conforms to this Standard 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 is provided as specified.

An implementation conforming to this International Standard may provide additional attributes, library units, and pragmas. However, it shall not provide any attribute, library unit, or pragma having the same name as an 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.

Documentation Requirements

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 Annex M.

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.

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.

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.

NOTES
2 The above requirements imply that an implementation conforming to this Standard may support some of the capabilities required by a Specialized Needs Annex without supporting all required capabilities.

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.

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.

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

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

  case_statement

- Boldface words are used to denote reserved words, for example:

  array

Withdrawn

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

  return_statement ::= **return** [expression];
  return_statement ::= **return**; | **return** expression;

- 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.

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

- A vertical line separates alternative items unless it occurs immediately after an opening curly bracket, in which case it stands for itself:

  constraint ::= scalar_constraint | composite_constraint
  discrete_choice_list ::= discrete_choice {| discrete_choice}

- 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.

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.

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

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

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 run-time 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.

NOTES

3 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:

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

4 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.

Withdrawn

1.1.5 Classification of Errors

Implementation Requirements

The language definition classifies errors into several different categories:

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

  These errors correspond to any violation of a rule given in this International Standard, 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.

  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.

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

  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;

  The language rules define certain kinds of errors that need not 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.

- Erroneous execution.

  In addition to bounded errors, the language rules define certain kinds of errors as leading to *erroneous execution*. Like bounded errors, the implementation need not 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.

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 International Standard; in particular, in the standard mode, all legal compilation_units shall be accepted.

Implementation Advice

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

Withdrawn

1.2 Normative References

The following standards contain provisions which, through reference in this text, constitute provisions of this International Standard. At the time of publication, the editions indicated were valid. All standards are subject to revision, and parties to agreements based on this International Standard are encouraged to investigate the possibility of applying the most recent editions of the standards indicated below. Members of IEC and ISO maintain registers of currently valid International Standards.

ISO/IEC 646:1991, *Information technology — ISO 7-bit coded character set for information interchange.*

ISO/IEC 1539:1991, *Information technology — Programming languages — FORTRAN.*

ISO 1989:1985, *Programming languages — COBOL.*

ISO/IEC 6429:1992, *Information technology — Control functions for coded graphic character sets.*

ISO/IEC 8859-1:1987, *Information processing — 8-bit single-byte coded character sets — Part 1: Latin alphabet No. 1.*

ISO/IEC 9899:1990, *Programming languages — C.*

ISO/IEC 10646-1:1993, *Information technology — Universal Multiple-Octet Coded Character Set (UCS) — Part 1: Architecture and Basic Multilingual Plane.*

Withdrawn