
British Standard Memorandum on the Standardization of

The programming language Ada

Mémoire sur la normalisation du langage de programmation Ada

Memorandum zur Normung der Programmiersprache Ada

National foreword

This British Standard has been prepared under the direction of the Information Systems Technology Standards Policy Committee. It implements the European Standard EN 28 652 : 1989 — Programming languages. Ada.

1. Scope

This British Standard records the international and European agreements that standardize the programming language Ada.

2. Memorandum

The international standard, ISO 8652 : 1987, published by the International Organization for Standardization (ISO), endorsed ANSI/MIL standard 1815A — 1983 as an international standard.

In December 1988, the European Committee for Standardization (CEN) adopted ISO 8652 : 1987 as a European Standard.

This British Standard constitutes the UK implementation of the European (and the international) standard.

NOTE. With this British Standard is sold a copy of ANSI/MIL 1815A — 1983.

This European Standard EN 28 652 has the status of a British Standard

Brief history

The Technical Board has decided to submit the International Standard ISO 8652, 1st edition, 1987 "Programming languages, Ada" to the Formal Vote. The result of this vote was positive.

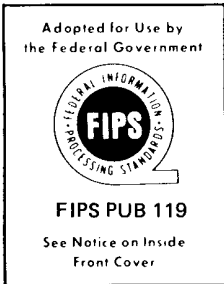
For the time being, this document exists only in the English and the French versions.

According to the CEN/CENELEC Common Rules, the following countries are bound to implement this standard : Austria, Belgium, Denmark, Finland, France, Germany, Greece, Iceland, Ireland, Italy, Luxemburg, Netherlands, Norway, Portugal, Spain, Sweden, Switzerland and United Kingdom.

Statement

The text of the International Standard ISO 8652, 1st edition, 1987, was approved by CEN as a European Standard without any modification.

American National Standard



reference manual for the Ada programming language



american national standards institute, inc.
1430 broadway, new york, new york 10018

American National Standard

An American National Standard implies a consensus of those substantially concerned with its scope and provisions. An American National Standard is intended as a guide to aid the manufacturer, the consumer, and the general public. The existence of an American National Standard does not in any respect preclude anyone, whether he has approved the standard or not, from manufacturing, marketing, purchasing, or using products, processes, or procedures not conforming to the standard. American National Standards are subject to periodic review and users are cautioned to obtain the latest editions.

The American National Standards Institute does not develop standards and will in no circumstances give an interpretation of any American National Standard. Moreover, no person shall have the right or authority to issue an interpretation of an American National Standard in the name of the American National Standards Institute.

CAUTION NOTICE: This American National Standard may be revised or withdrawn at any time. The procedures of the American National Standards Institute require that action be taken to reaffirm, revise, or withdraw this standard no later than five years from the date of approval. Purchasers of American National Standards may receive current information on all standards by calling or writing the American National Standards Institute.

This standard has been adopted for Federal Government use.

Details concerning its use within the Federal Government are contained in Federal Information Processing Standards Publication 119, ADA. For a complete list of the publications available in the Federal Information Processing Standards Series, write to the Standards Processing Coordinator (ADP), Institute for Computer Sciences and Technology, National Bureau of Standards, Gaithersburg, MD 20899.

American National Standard Reference Manual for the Ada[®] Programming Language

Sponsor

United States Department of Defense

Approved February 17, 1983

American National Standards Institute, Inc

®Ada is a registered trademark of the U.S. Government, Ada Joint Program Office

Note (The information on this page is not a part of American National Standard Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A-1983.)

Ada[®] is a registered trademark of the United States Government, Department of Defense, Under Secretary for Research and Engineering. Its use is administered by the Ada Joint Program Office (AJPO). In all contexts, use of the term "Ada" should indicate conformance to the standard. In keeping with policies on voluntary conformance, use of the term Ada is equivalent to a voluntary statement of conformance to the standard.

The use of the trademarked term Ada will be made freely available to those who use it to indicate conformance to the standard and in accordance with the following guidelines:

In any published material the first appearance of the term Ada must be properly acknowledged and include the statement "Ada is a registered trademark of the U.S. Government (Ada Joint Program Office)."

Describing, advertising, or promoting a language processor as an "Ada" processor is equivalent to making a voluntary statement of conformance to ANSI/MIL-STD-1815A.

The term Ada may be used in describing language processors which are not completely conforming or are not making a claim of conformance provided that there is a precise, easily visible statement of their non-conformance at the same time and in the same context.

Uses of the term Ada other than those described above, including all organizations, companies and product names incorporating or utilizing the term Ada, need written authorization from the AJPO. Those persons advertising or otherwise promoting a language processor asserted as being a standard Ada processor for sale or public use are required to provide the AJPO with evidence sufficient to demonstrate conformance to the Ada standard.

Use of the trademark does not imply any endorsement or warranty of the product by either DoD or ANSI.

The Department of Defense (DoD), as the trademark owner, will allow others to use the Ada trademark free of charge and will not take action to prevent use of the Ada trademark so long as the trademark is used properly according to the above policy. Misuse of the trademark may lead to legal action.

In the interest of information interchange, all users of this standard are encouraged to contact the Ada Joint Program Office, Department of Defense, OUSD(R&E), Washington, D.C. 20301, U.S.A. Users of the trademark and those reprinting the standard are required to notify the AJPO.

Foreword

(This Foreword is not a part of American National Standard Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A-1983.)

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

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.

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.

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.

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.

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.

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, AdaTEC, Afcet, those of the LMSC review team, and those of the Ada Tokyo Study Group.

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.

This 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 and Robert F. Mathis, served as the technical representatives of the United States Government and effectively coordinated the efforts of all participants in the Ada program.

This reference manual was prepared with a formatter specialized for Ada texts. It was developed by Jon F. Hueras for Multics, using the Cii Honeywell Bull photocomposition system.

Table of Contents

1. Introduction		
1.1	Scope of the Standard	1- 1
1.1.1	Extent of the Standard	1- 1
1.1.2	Conformity of an Implementation with the Standard	1- 2
1.2	Structure of the Standard	1- 3
1.3	Design Goals and Sources	1- 3
1.4	Language Summary	1- 4
1.5	Method of Description and Syntax Notation	1- 7
1.6	Classification of Errors	1- 8
2. Lexical Elements		
2.1	Character Set	2- 1
2.2	Lexical Elements, Separators, and Delimiters	2- 2
2.3	Identifiers	2- 4
2.4	Numeric Literals	2- 4
2.4.1	Decimal Literals	2- 4
2.4.2	Based Literals	2- 5
2.5	Character Literals	2- 6
2.6	String Literals	2- 6
2.7	Comments	2- 7
2.8	Pragmas	2- 7
2.9	Reserved Words	2- 9
2.10	Allowable Replacements of Characters	2- 9
3. Declarations and Types		
3.1	Declarations	3- 1
3.2	Objects and Named Numbers	3- 2
3.2.1	Object Declarations	3- 3
3.2.2	Number Declarations	3- 5
3.3	Types and Subtypes	3- 6
3.3.1	Type Declarations	3- 7
3.3.2	Subtype Declarations	3- 8
3.3.3	Classification of Operations	3- 9
3.4	Derived Types	3-10
3.5	Scalar Types	3-12
3.5.1	Enumeration Types	3-13
3.5.2	Character Types	3-14
3.5.3	Boolean Types	3-15
3.5.4	Integer Types	3-15
3.5.5	Operations of Discrete Types	3-16
3.5.6	Real Types	3-19
3.5.7	Floating Point Types	3-20
3.5.8	Operations of Floating Point Types	3-22
3.5.9	Fixed Point Types	3-24
3.5.10	Operations of Fixed Point Types	3-26
3.6	Array Types	3-27
3.6.1	Index Constraints and Discrete Ranges	3-29
3.6.2	Operations of Array Types	3-31
3.6.3	The Type String	3-32
3.7	Record Types	3-33
3.7.1	Discriminants	3-34
3.7.2	Discriminant Constraints	3-36
3.7.3	Variant Parts	3-38
3.7.4	Operations of Record Types	3-39
3.8	Access Types	3-40
3.8.1	Incomplete Type Declarations	3-41
3.8.2	Operations of Access Types	3-42
3.9	Declarative Parts	3-43

4. Names and Expressions	
4.1 Names	4- 1
4.1.1 Indexed Components	4- 2
4.1.2 Slices	4- 3
4.1.3 Selected Components	4- 3
4.1.4 Attributes	4- 5
4.2 Literals	4- 6
4.3 Aggregates	4- 7
4.3.1 Record Aggregates	4- 8
4.3.2 Array Aggregates	4- 9
4.4 Expressions	4-11
4.5 Operators and Expression Evaluation	4-12
4.5.1 Logical Operators and Short-circuit Control Forms	4-13
4.5.2 Relational Operators and Membership Tests	4-14
4.5.3 Binary Adding Operators	4-16
4.5.4 Unary Adding Operators	4-16
4.5.5 Multiplying Operators	4-17
4.5.6 Highest Precedence Operators	4-19
4.5.7 Accuracy of Operations with Real Operands	4-20
4.6 Type Conversions	4-21
4.7 Qualified Expressions	4-24
4.8 Allocators	4-24
4.9 Static Expressions and Static Subtypes	4-26
4.10 Universal Expressions	4-27
 5. Statements	
5.1 Simple and Compound Statements - Sequences of Statements	5- 1
5.2 Assignment Statement	5- 2
5.2.1 Array Assignments	5- 3
5.3 If Statements	5- 4
5.4 Case Statements	5- 5
5.5 Loop Statements	5- 7
5.6 Block Statements	5- 9
5.7 Exit Statements	5-10
5.8 Return Statements	5-10
5.9 Goto Statements	5-11
 6. Subprograms	
6.1 Subprogram Declarations	6- 1
6.2 Formal Parameter Modes	6- 3
6.3 Subprogram Bodies	6- 4
6.3.1 Conformance Rules	6- 5
6.3.2 Inline Expansion of Subprograms	6- 6
6.4 Subprogram Calls	6- 7
6.4.1 Parameter Associations	6- 8
6.4.2 Default Parameters	6- 9
6.5 Function Subprograms	6-10
6.6 Parameter and Result Type Profile - Overloading of Subprograms	6-10
6.7 Overloading of Operators	6-11
 7. Packages	
7.1 Package Structure	7- 1
7.2 Package Specifications and Declarations	7- 2
7.3 Package Bodies	7- 3
7.4 Private Type and Deferred Constant Declarations	7- 5
7.4.1 Private Types	7- 5
7.4.2 Operations of a Private Type	7- 6
7.4.3 Deferred Constants	7- 9
7.4.4 Limited Types	7- 9
7.5 Example of a Table Management Package	7-11
7.6 Example of a Text Handling Package	7-12

8. Visibility Rules	
8.1	Declarative Region 8- 1
8.2	Scope of Declarations 8- 2
8.3	Visibility 8- 3
8.4	Use Clauses 8- 6
8.5	Renaming Declarations 8- 8
8.6	The Package Standard 8-10
8.7	The Context of Overload Resolution 8-11
9. Tasks	
9.1	Task Specifications and Task Bodies 9- 1
9.2	Task Types and Task Objects 9- 3
9.3	Task Execution - Task Activation 9- 5
9.4	Task Dependence - Termination of Tasks 9- 6
9.5	Entries, Entry Calls, and Accept Statements 9- 8
9.6	Delay Statements, Duration, and Time 9-10
9.7	Select Statements 9-12
9.7.1	Selective Waits 9-12
9.7.2	Conditional Entry Calls 9-14
9.7.3	Timed Entry Calls 9-15
9.8	Priorities 9-16
9.9	Task and Entry Attributes 9-17
9.10	Abort Statements 9-18
9.11	Shared Variables 9-19
9.12	Example of Tasking 9-20
10. Program Structure and Compilation Issues	
10.1	Compilation Units - Library Units 10- 1
10.1.1	Context Clauses - With Clauses 10- 2
10.1.2	Examples of Compilation Units 10- 4
10.2	Subunits of Compilation Units 10- 6
10.2.1	Examples of Subunits 10- 7
10.3	Order of Compilation 10- 9
10.4	The Program Library 10-11
10.5	Elaboration of Library Units 10-11
10.6	Program Optimization 10-12
11. Exceptions	
11.1	Exception Declarations 11- 1
11.2	Exception Handlers 11- 3
11.3	Raise Statements 11- 4
11.4	Exception Handling 11- 4
11.4.1	Exceptions Raised During the Execution of Statements 11- 4
11.4.2	Exceptions Raised During the Elaboration of Declarations 11- 7
11.5	Exceptions Raised During Task Communication 11- 8
11.6	Exceptions and Optimization 11- 9
11.7	Suppressing Checks 11-10

12. Generic Units	
12.1	Generic Declarations 12- 1
12.1.1	Generic Formal Objects 12- 3
12.1.2	Generic Formal Types 12- 4
12.1.3	Generic Formal Subprograms 12- 6
12.2	Generic Bodies 12- 6
12.3	Generic Instantiation 12- 8
12.3.1	Matching Rules for Formal Objects 12-10
12.3.2	Matching Rules for Formal Private Types 12-11
12.3.3	Matching Rules for Formal Scalar Types 12-11
12.3.4	Matching Rules for Formal Array Types 12-12
12.3.5	Matching Rules for Formal Access Types 12-13
12.3.6	Matching Rules for Formal Subprograms 12-14
12.4	Example of a Generic Package 12-15
13. Representation Clauses and Implementation-Dependent Features	
13.1	Representation Clauses 13- 1
13.2	Length Clauses 13- 3
13.3	Enumeration Representation Clauses 13- 5
13.4	Record Representation Clauses 13- 5
13.5	Address Clauses 13- 7
13.5.1	Interrupts 13- 8
13.6	Change of Representation 13- 9
13.7	The Package System 13- 9
13.7.1	System-Dependent Named Numbers 13-11
13.7.2	Representation Attributes 13-12
13.7.3	Representation Attributes of Real Types 13-13
13.8	Machine Code Insertions 13-14
13.9	Interface to Other Languages 13-15
13.10	Unchecked Programming 13-16
13.10.1	Unchecked Storage Deallocation 13-16
13.10.2	Unchecked Type Conversions 13-17
14. Input-Output	
14.1	External Files and File Objects 14- 1
14.2	Sequential and Direct Files 14- 2
14.2.1	File Management 14- 3
14.2.2	Sequential Input-Output 14- 5
14.2.3	Specification of the Package Sequential_IO 14- 6
14.2.4	Direct Input- Output 14- 7
14.2.5	Specification of the Package Direct_IO 14- 8
14.3	Text Input- Output 14- 9
14.3.1	File Management 14-11
14.3.2	Default Input and Output Files 14-11
14.3.3	Specification of Line and Page Lengths 14-12
14.3.4	Operations on Columns, Lines, and Pages 14-13
14.3.5	Get and Put Procedures 14-17
14.3.6	Input-Output of Characters and Strings 14-19
14.3.7	Input-Output for Integer Types 14-20
14.3.8	Input-Output for Real Types 14-22
14.3.9	Input-Output for Enumeration Types 14-24
14.3.10	Specification of the Package Text_IO 14-26
14.4	Exceptions in Input-Output 14-30
14.5	Specification of the Package IO_Exceptions 14-32
14.6	Low Level Input-Output 14-32
14.7	Example of Input-Output 14-33

Annexes

- A. Predefined Language Attributes
- B. Predefined Language Pragmas
- C. Predefined Language Environment

Appendices

- D. Glossary
- E. Syntax Summary
- F. Implementation-Dependent Characteristics

Index

Postscript: Submission of Comments

1. Introduction

Ada is a programming language designed in accordance with requirements defined by the United States Department of Defense: the so-called Steelman requirements. Overall, these requirements call for a language with considerable expressive power covering a wide application domain. As a result, the language includes facilities offered by classical languages such as Pascal as well as facilities often found only in specialized languages. Thus the language is a modern algorithmic language with the usual control structures, and with the ability to define types and subprograms. It also serves the need for modularity, whereby data, types, and subprograms can be packaged. It treats modularity in the physical sense as well, with a facility to support separate compilation.

In addition to these aspects, the language covers real-time programming, with facilities to model parallel tasks and to handle exceptions. It also covers systems programming; this requires precise control over the representation of data and access to system-dependent properties. Finally, both application-level and machine-level input-output are defined.

1.1 Scope of the Standard

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

1.1.1 Extent of the Standard

This standard specifies:

- (a) The form of a program unit written in Ada.
- (b) The effect of translating and executing such a program unit.
- (c) The manner in which program units may be combined to form Ada programs.
- (d) The predefined program units that a conforming implementation must supply.
- (e) The permissible variations within the standard, and the manner in which they must be specified.
- (f) Those violations of the standard that a conforming implementation is required to detect, and the effect of attempting to translate or execute a program unit containing such violations.
- (g) Those violations of the standard that a conforming implementation is not required to detect.

This standard does not specify:

- (h) The means whereby a program unit written in Ada is transformed into object code executable by a processor.
- (i) The means whereby translation or execution of program units is invoked and the executing units are controlled.
- (j) The size or speed of the object code, or the relative execution speed of different language constructs.
- (k) The form or contents of any listings produced by implementations; in particular, the form or contents of error or warning messages.
- (l) The effect of executing a program unit that contains any violation that a conforming implementation is not required to detect.
- (m) The size of a program or program unit that will exceed the capacity of a particular conforming implementation.

Where this standard specifies that a program unit written in Ada has an exact effect, this effect is the operational meaning of the program unit and must be produced by all conforming implementations. Where this standard specifies permissible variations in the effects of constituents of a program unit written in Ada, the operational meaning of the program unit as a whole is understood to be the range of possible effects that result from all these variations, and a conforming implementation is allowed to produce any of these possible effects. Examples of permissible variations are:

- The represented values of fixed or floating numeric quantities, and the results of operations upon them.
- The order of execution of statements in different parallel tasks, in the absence of explicit synchronization.

1.1.2 Conformity of an Implementation with the Standard

A conforming implementation is one that:

- (a) Correctly translates and executes legal program units written in Ada, provided that they are not so large as to exceed the capacity of the implementation.
- (b) Rejects all program units that are so large as to exceed the capacity of the implementation.
- (c) Rejects all program units that contain errors whose detection is required by the standard.
- (d) Supplies all predefined program units required by the standard.
- (e) Contains no variations except where the standard permits.
- (f) Specifies all such permitted variations in the manner prescribed by the standard.

1.1.2 Conformity of an Implementation with the Standard

1-2

1.2 Structure of the Standard

This reference manual contains fourteen chapters, three annexes, three appendices, and an index. 1

Each chapter is divided into sections that have a common structure. Each section introduces its subject, gives any necessary syntax rules, and describes the semantics of the corresponding language constructs. Examples and notes, and then references, may appear at the end of a section. 2

Examples are meant to illustrate the possible forms of the constructs described. Notes are meant to emphasize consequences of the rules described in the section or elsewhere. References are meant to attract the attention of readers to a term or phrase having a technical meaning defined in another section. 3

The standard definition of the Ada programming language consists of the fourteen chapters and the three annexes, subject to the following restriction: the material in each of the items listed below is informative, and not part of the standard definition of the Ada programming language: 4

- Section 1.3 Design goals and sources 5
- Section 1.4 Language summary 6
- The examples, notes, and references given at the end of each section 7
- Each section whose title starts with the word "Example" or "Examples" 8

1.3 Design Goals and Sources

Ada was designed with three overriding concerns: program reliability and maintenance, programming as a human activity, and efficiency. 1

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. 2

Concern for the human programmer was also stressed during the design. Above all, an attempt was made to keep the language as small as possible, given the ambitious nature of the application domain. We have attempted to cover this domain with a small number of underlying concepts integrated in a consistent and systematic way. Nevertheless we have tried to avoid the pitfalls of excessive involution, and in the constant search for simpler designs we have tried to provide language constructs that correspond intuitively to what the users will normally expect. 3

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 has been a central idea in this 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. 4

- 5 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.
- 6 None of the above design goals was considered as achievable after the fact. The design goals drove the entire design process from the beginning.
- 7 A perpetual difficulty in language design is that one must both identify the capabilities required by the application domain and design language features that provide these capabilities. The difficulty existed in this design, although to a lesser degree than usual because of the Steelman requirements. These requirements often simplified the design process by allowing it to concentrate on the design of a given system providing a well defined set of capabilities, rather than on the definition of the capabilities themselves.
- 8 Another significant simplification of the design work resulted from earlier experience acquired by several successful Pascal derivatives developed with similar goals. These are the languages Euclid, Lis, Mesa, Modula, and Sue. Many of the key ideas and syntactic forms developed in these languages have counterparts in Ada. Several existing languages such as Algol 68 and Simula, and also recent research languages such as Alphard and Clu, influenced this language in several respects, although to a lesser degree than did the Pascal family.
- 9 Finally, the evaluation reports received on an earlier formulation (the Green language), and on alternative proposals (the Red, Blue, and Yellow languages), the language reviews that took place at different stages of this project, and the thousands of comments received from fifteen different countries during the preliminary stages of the Ada design and during the ANSI canvass, all had a significant impact on the standard definition of the language.

1.4 Language Summary

- 1 An Ada program is composed of one or more program units. These program units can be compiled separately. Program units may be subprograms (which define executable algorithms), package units (which define collections of entities), task units (which define parallel computations), or generic units (which define parameterized forms of packages and subprograms). Each 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.
- 2 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.
- 3 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. The text of a separately compiled program unit must name the library units it requires.

4 *Program Units*

- 5 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 common pool of data and types, a collection of related subprograms, or 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.

A task unit is the basic unit for defining a task whose sequence of actions may be executed in parallel 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.

Declarations and Statements

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, 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 an exit, return, or goto statement, or the raising of an exception, 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 only applicable to tasks. A delay statement delays the execution of a task for a specified duration. An entry call statement is written as a procedure call statement; it specifies that the task issuing the call is ready for a rendezvous with another task that has this entry. The called task is ready to accept the entry call when its execution reaches a corresponding accept statement, which specifies the actions then to be performed. After completion of the rendezvous, both the calling task and the task having the entry may continue their execution in parallel. 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.

18 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.

19 *Data Types*

20 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 scalar types (comprising enumeration and numeric types), composite types, access types, and private types.

21 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 and CHARACTER are predefined.

22 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.

23 Composite types allow definitions of structured objects with related components. The composite types in the language provide for 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. The array type STRING is predefined.

24 A record may have special components called discriminants. Alternative record structures that depend on the values of discriminants can be defined within a record type.

25 Access types allow the construction of linked data structures created by the evaluation of allocators. They allow several variables of an access type to designate the same object, and components of one object to designate the same or other objects. Both the elements in such a linked data structure and their relation to other elements can be altered during program execution.

26 Private types can be defined in a package that conceals structural details that are externally irrelevant. Only the logically necessary properties (including any discriminants) are made visible to the users of such types.

27 The concept of a type is 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.

28 *Other Facilities*

29 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.

30 Input-output is defined in the language by means of predefined library packages. Facilities are provided for input-output of values of user-defined as well as of predefined types. Standard means of representing values in display form are also provided.

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 so allow general algorithms to be applied to all types of a given class.

31

1.5 Method of Description and Syntax Notation

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

1

The meaning of Ada program units is described by means of narrative rules defining both the effects of each construct and the composition rules for constructs. This narrative employs technical terms whose precise definition is given in the text (references to the section containing the definition of a technical term appear at the end of each section that uses the term).

2

All other terms are in the English language and bear their natural meaning, as defined in Webster's Third New International Dictionary of the English Language.

3

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

4

- (a) Lower case words, some containing embedded underlines, are used to denote syntactic categories, for example:

5

`adding_operator`

Whenever the name of a syntactic category is used apart from the syntax rules themselves, spaces take the place of the underlines (thus: `adding operator`).

6

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

7

array

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

8

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

- (d) Braces 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.

9

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

- 10 (e) A vertical bar separates alternative items unless it occurs immediately after an opening brace, in which case it stands for itself:

```
letter_or_digit ::= letter | digit
component_association ::= [choice || choice] => expression
```

- 11 (f) 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 *type_name* and *task_name* are both equivalent to name alone.

Note:

- 12 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;
```

- 13 Different lines are used for parts of a syntax rule if the corresponding parts of the construct described by the rule are intended to be on different lines. Indentation in the rule is a recommendation for indentation of the corresponding part of the construct. It is recommended that all indentations be by multiples of a basic step of indentation (the number of spaces for the basic step is not defined). The preferred places for other line breaks are after semicolons. On the other hand, if a complete construct can fit on one line, this is also allowed in the recommended paragraphing.

1.6 Classification of Errors

- 1 The language definition classifies errors into several different categories:

- 2 (a) Errors that must be detected at compilation time by every Ada compiler.

3 These errors correspond to any violation of a rule given in this reference manual, other than the violations that correspond to (b) or (c) below. In particular, violation of any rule that uses the terms *must*, *allowed*, *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 (b) Errors that must be detected at run time by the execution of an Ada program.

5 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 an exception is certain to be raised in every execution of a program, then compilers are allowed (although not required) to report this fact at compilation time.

(c) Erroneous execution.

6

The language rules specify certain rules to be obeyed by Ada programs, although there is no requirement on Ada compilers to provide either a compilation-time or a run-time detection of the violation of such rules. The errors of this category are indicated by the use of the word *erroneous* to qualify the execution of the corresponding constructs. The effect of erroneous execution is unpredictable.

7

(d) Incorrect order dependences.

8

Whenever the reference manual specifies that different parts of a given construct are to be executed *in some order that is not defined by the language*, this means that the implementation is allowed to execute these parts in any given order, following the rules that result from that given order, but not in parallel. Furthermore, the construct is incorrect if execution of these parts in a different order would have a different effect. Compilers are not required to provide either compilation-time or run-time detection of incorrect order dependences. The foregoing is expressed in terms of the process that is called execution; it applies equally to the processes that are called evaluation and elaboration.

9

If a compiler is able to recognize at compilation time that a construct is erroneous or contains an incorrect order dependence, then the compiler is allowed to generate, in place of the code otherwise generated for the construct, code that raises the predefined exception `PROGRAM_ERROR`. Similarly, compilers are allowed to generate code that checks at run time for erroneous constructs, for incorrect order dependences, or for both. The predefined exception `PROGRAM_ERROR` is raised if such a check fails.

10

2. Lexical Elements

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 chapter. Pragmas, which provide certain information for the compiler, are also described in this chapter.

References: character 2.1, compilation 10.1, lexical element 2.2, pragma 2.8

2.1 Character Set

The only characters allowed in the text of a program are the graphic characters and format effectors. Each graphic character corresponds to a unique code of the *ISO* seven-bit coded character set (*ISO* standard 646), and is represented (visually) by a graphical symbol. Some graphic characters are represented by different graphical symbols in alternative national representations of the *ISO* character set. The description of the language definition in this standard reference manual uses the *ASCII* graphical symbols, the *ANSI* graphical representation of the *ISO* character set.

```
graphic_character ::= basic_graphic_character  
                  | lower_case_letter | other_special_character
```

```
basic_graphic_character ::=  
    upper_case_letter | digit  
    | special_character | space_character
```

```
basic_character ::=  
    basic_graphic_character | format_effector
```

The basic character set is sufficient for writing any program. The characters included in each of the categories of basic graphic characters are defined as follows:

(a) upper case letters

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

(b) digits

0 1 2 3 4 5 6 7 8 9

(c) special characters

" # & ' () * + , - . / : ; < = > _ |

(d) the space character

Format effectors are the *ISO* (and *ASCII*) characters called horizontal tabulation, vertical tabulation, carriage return, line feed, and form feed.

9 The characters included in each of the remaining categories of graphic characters are defined as follows:

10 (e) lower case letters
a b c d e f g h i j k l m n o p q r s t u v w x y z

11 (f) other special characters
! \$ % ? @ [\] ^ ` { } ~

12 Allowable replacements for the special characters vertical bar (|), sharp (≠), and quotation (") are defined in section 2.10.

Notes:

13 The *ISO* character that corresponds to the sharp graphical symbol in the *ASCII* representation appears as a pound sterling symbol in the French, German, and United Kingdom standard national representations. In any case, the font design of graphical symbols (for example, whether they are in italic or bold typeface) is not part of the *ISO* standard.

14 The meanings of the acronyms used in this section are as follows: *ANSI* stands for American National Standards Institute, *ASCII* stands for American Standard Code for Information Interchange, and *ISO* stands for International Organization for Standardization.

15 The following names are used when referring to special characters and other special characters:

<i>symbol</i>	<i>name</i>	<i>symbol</i>	<i>name</i>
"	quotation	>	greater than
≠	sharp	—	underline
&	ampersand		vertical bar
'	apostrophe	!	exclamation mark
(	left parenthesis	\$	dollar
)	right parenthesis	%	percent
*	star, multiply	?	question mark
+	plus	@	commercial at
,	comma	[	left square bracket
-	hyphen, minus	\	back-slash
.	dot, point, period	]	right square bracket
/	slash, divide	^	circumflex
:	colon	`	grave accent
;	semicolon	{	left brace
<	less than	}	right brace
=	equal	~	tilde

2.2 Lexical Elements, Separators, and Delimiters

1 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 either a delimiter, an identifier (which may be a reserved word), a numeric literal, a character literal, a string literal, or a comment. The effect of a program depends only on the particular sequences of lexical elements that form its compilations, excluding the comments, if any.

In some cases an explicit *separator* is required to separate adjacent lexical elements (namely, when without separation, interpretation as a single lexical element is possible). A separator is any of a space character, a format effector, or the end of a line. A space character is a separator except within a comment, a string literal, or a space character literal. Format effectors other than horizontal tabulation are always separators. Horizontal tabulation is a separator except within a comment.

The end of a line is always a separator. The language does not define what causes the end of a line. However if, for a given implementation, the end of a line is signified by one or more characters, then these characters must be format effectors other than horizontal tabulation. In any case, a sequence of one or more format effectors other than horizontal tabulation must cause at least one end of line.

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 or a numeric literal and an adjacent identifier or numeric literal.

A *delimiter* is either one of the following special characters (in the basic character set)

& ' () * + , - . / : ; < = > |

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

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

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.

The remaining forms of lexical element are described in other sections of this chapter.

Notes:

Each lexical element must fit on one line, since the end of a line is a separator. The quotation, sharp, and underline characters, likewise two adjacent hyphens, are not delimiters, but may form part of other lexical elements.

The following names are used when referring to compound delimiters:

<i>delimiter</i>	<i>name</i>
=>	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

References: character literal 2.5, comment 2.7, compilation 10.1, format effector 2.1, identifier 2.3, numeric literal 2.4, reserved word 2.9, space character 2.1, special character 2.1, string literal 2.6

2.3 Identifiers

Identifiers are used as names and also as reserved words.

```

identifier ::=
    letter {[underline] letter_or_digit}

letter_or_digit ::= letter | digit

letter ::= upper_case_letter | lower_case_letter

```

All characters of an identifier are significant, including any underline character inserted between a letter or digit and an adjacent letter or digit. Identifiers differing only in the use of corresponding upper and lower case letters are considered as the same.

Examples:

```

COUNT      X      get_symbol      Ethelyn      Marion
SNOBOL_4    X1     PageCount      STORE_NEXT_ITEM

```

Note:

No space is allowed within an identifier since a space is a separator.

References: digit 2.1, lower case letter 2.1, name 4.1, reserved word 2.9, separator 2.2, space character 2.1, upper case letter 2.1

2.4 Numeric Literals

There are two classes 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. Real literals are the literals of the type *universal_real*. Integer literals are the literals of the type *universal_integer*.

```

numeric_literal ::= decimal_literal | based_literal

```

References: literal 4.2, universal_integer type 3.5.4, universal_real type 3.5.6

2.4.1 Decimal Literals

A decimal literal is a numeric literal expressed in the conventional decimal notation (that is, the base is implicitly ten).

```

decimal_literal ::= integer [.integer] [exponent]

integer ::= digit {[underline] digit}

exponent ::= E [+] integer | E - integer

```

2.4.1 Decimal Literals

2-4

An underline character inserted between adjacent digits of a decimal literal does not affect the value of this numeric literal. The letter E of the exponent, if any, can be written either in lower case or in upper case, with the same meaning. 3

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. An exponent for an integer literal must not have a minus sign. 4

Examples: 5

```
12      0      1E6    123_456    -- integer literals
12.0    0.0    0.456  3.14159_26 -- real literals
1.34E-12 1.0E+6 -- real literals with exponent
```

Notes:

Leading zeros are allowed. No space is allowed in a numeric literal, not even between constituents of the exponent, since a space is a separator. A zero exponent is allowed for an integer literal. 6

References: digit 2.1, lower case letter 2.1, numeric literal 2.4, separator 2.2, space character 2.1, upper case letter 2.1 7

2.4.2 Based Literals

A based literal is a numeric literal expressed in a form that specifies the base explicitly. The base must be at least two and at most sixteen. 1

```
based_literal ::=
    base # based_integer [.based_integer] # [exponent] 2

base ::= integer

based_integer ::=
    extended_digit {[underline] extended_digit}

extended_digit ::= digit | letter
```

An underline character inserted between adjacent digits of a based literal does not affect the value of this numeric literal. The base and the exponent, if any, are in decimal notation. The only letters allowed as extended digits are the letters A through F for the digits ten through fifteen. A letter in a based literal (either an extended digit or the letter E of an exponent) can be written either in lower case or in upper case, with the same meaning. 3

The conventional meaning of based notation is assumed; in particular the value of each extended digit of a based literal must be less than the base. 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. 4

Examples:

2#1111_1111#	16#FF#	016#OFF#	-- 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_111#E11		-- real literals of value 4095.0

References: digit 2.1, exponent 2.4.1, letter 2.3, lower case letter 2.1, numeric literal 2.4, upper case letter 2.1

2.5 Character Literals

A character literal is formed by enclosing one of the 95 graphic characters (including the space) between two apostrophe characters. A character literal has a value that belongs to a character type.

character_literal ::= 'graphic_character'

Examples:

'A' '*' ''' ' '

References: character type 3.5.2, graphic character 2.1, literal 4.2, space character 2.1

2.6 String Literals

A string literal is formed by a sequence of graphic characters (possibly none) enclosed between two quotation characters used as *string brackets*.

string_literal ::= "{graphic_character}"

A string literal has a value that is a sequence of character values corresponding to the graphic characters of the string literal apart from the quotation character itself. If a quotation character value is to be represented in the sequence of character values, then a pair of adjacent quotation characters must be written at the corresponding place within the string literal. (This means that a string literal that includes two adjacent quotation characters is never interpreted as two adjacent string literals.)

The *length* of a string literal is the number of character values in the sequence represented. (Each doubled quotation character is counted as a single character.)

Examples:

"Message of the day:"

""	-- an empty string literal
" " "A" "****"	-- three string literals of length 1

"Characters such as \$, %, and } are allowed in string literals"

Note:

A string literal must fit on one line since it is a lexical element (see 2.2). Longer sequences of graphic character values can be obtained by catenation of string literals. Similarly catenation of constants declared in the package `ASCII` can be used to obtain sequences of character values that include nongraphic character values (the so-called control characters). Examples of such uses of catenation are given below:

```
"FIRST PART OF A SEQUENCE OF CHARACTERS " &
"THAT CONTINUES ON THE NEXT LINE"
```

```
"sequence that includes the" & ASCII.ACK & "control character"
```

References: `ascii` predefined package C, catenation operation 4.5.3, character value 3.5.2, constant 3.2.1, declaration 3.1, end of a line 2.2, graphic character 2.1, lexical element 2.2

2.7 Comments

A comment starts with two adjacent hyphens and extends up to the end of the line. A comment can appear on any line of a program. The presence or absence of comments has no influence on whether a program is legal or illegal. Furthermore, comments do not influence the effect of a program; their sole purpose is the enlightenment of the human reader.

Examples:

```
-- the last sentence above echoes the Algol 68 report

end; -- processing of LINE is complete

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

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

Note:

Horizontal tabulation can be used in comments, after the double hyphen, and is equivalent to one or more spaces (see 2.2).

References: end of a line 2.2, illegal 1.6, legal 1.6, space character 2.1

2.8 Pragmas

A pragma is used to convey information to the compiler. A pragma starts with the reserved word **pragma** followed by an identifier that is the name of the pragma.

```
pragma ::=
  pragma identifier [(argument_association {, argument_association})];

argument_association ::=
  [argument_identifier =>] name
  | [argument_identifier =>] expression
```

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

- After a semicolon delimiter, but not within a formal part or discriminant part.
- At any place where the syntax rules allow a construct defined by a syntactic category whose name ends with "declaration", "statement", "clause", or "alternative", or one of the syntactic categories variant and exception handler; but not in place of such a construct. Also at any place where a compilation unit would be allowed.

Additional restrictions exist for the placement of specific pragmas.

Some pragmas have arguments. Argument associations can be either positional or named as for parameter associations of subprogram calls (see 6.4). Named associations are, however, only possible if the argument identifiers are defined. A name given in an argument must be either a name visible at the place of the pragma or an identifier specific to the pragma.

The pragmas defined by the language are described in Annex B: they must be supported by every implementation. In addition, an implementation may provide implementation-defined pragmas, which must then be described in Appendix F. An implementation is not allowed to define pragmas whose presence or absence influences the legality of the text outside such pragmas. Consequently, the legality of a program does not depend on the presence or absence of implementation-defined pragmas.

A pragma that is not language-defined has no effect if its identifier is not recognized by the (current) implementation. Furthermore, a pragma (whether language-defined or implementation-defined) has no effect if its placement or its arguments do not correspond to what is allowed for the pragma. The region of text over which a pragma has an effect depends on the pragma.

Examples:

```
pragma LIST(OFF);
pragma OPTIMIZE(TIME);
pragma INLINE(SETMASK);
pragma SUPPRESS(RANGE_CHECK, ON => INDEX);
```

Note:

It is recommended (but not required) that implementations issue warnings for pragmas that are not recognized and therefore ignored.

References: compilation unit 10.1, delimiter 2.2, discriminant part 3.7.1, exception handler 11.2, expression 4.4, formal part 6.1, identifier 2.3, implementation-defined pragma F, language-defined pragma B, legal 1.6, name 4.1, reserved word 2.9, statement 5, static expression 4.9, variant 3.7.3, visibility 8.3

Categories ending with "declaration" comprise: basic declaration 3.1, component declaration 3.7, entry declaration 9.5, generic parameter declaration 12.1

Categories ending with "clause" comprise: alignment clause 13.4, component clause 13.4, context clause 10.1.1, representation clause 13.1, use clause 8.4, with clause 10.1.1

Categories ending with "alternative" comprise: accept alternative 9.7.1, case statement alternative 5.4, delay alternative 9.7.1, select alternative 9.7.1, selective wait alternative 9.7.1, terminate alternative 9.7.1

2.9 Reserved Words

The identifiers listed below are called *reserved words* and are reserved for special significance in the language. For readability of this manual, the reserved words appear in lower case boldface.

abort	declare	generic	of	select
abs	delay	goto	or	separate
accept	delta		others	subtype
access	digits	if	out	
all	do	in		task
and		is	package	terminate
array			pragma	then
at	else		private	type
	elsif	limited	procedure	
	end	loop		
begin	entry		raise	use
body	exception		range	
	exit	mod	record	when
			rem	while
		new	renames	with
case	for	not	return	
constant	function	null	reverse	xor

A reserved word must not be used as a declared identifier.

Notes:

Reserved words differing only in the use of corresponding upper and lower case letters are considered as the same (see 2.3). In some attributes the identifier that appears after the apostrophe is identical to some reserved word.

References: attribute 4.1.4, declaration 3.1, identifier 2.3, lower case letter 2.1, upper case letter 2.1

2.10 Allowable Replacements of Characters

The following replacements are allowed for the vertical bar, sharp, and quotation basic characters:

- A vertical bar character (|) can be replaced by an exclamation mark (!) where used as a delimiter.
- The sharp characters (#) of a based literal can be replaced by colons (:) provided that the replacement is done for both occurrences.
- The quotation characters (") used as string brackets at both ends of a string literal can be replaced by percent characters (%) provided that the enclosed sequence of characters contains no quotation character, and provided that both string brackets are replaced. Any percent character within the sequence of characters must then be doubled and each such doubled percent character is interpreted as a single percent character value.

- 5 These replacements do not change the meaning of the program.

Notes:

- 6 It is recommended that use of the replacements for the vertical bar, sharp, and quotation characters be restricted to cases where the corresponding graphical symbols are not available. Note that the vertical bar appears as a broken bar on some equipment; replacement is not recommended in this case.
- 7 The rules given for identifiers and numeric literals are such that lower case and upper case letters can be used indifferently; these lexical elements can thus be written using only characters of the basic character set. If a string literal of the predefined type STRING contains characters that are not in the basic character set, the same sequence of character values can be obtained by concatenating string literals that contain only characters of the basic character set with suitable character constants declared in the predefined package ASCII. Thus the string literal "AB\$CD" could be replaced by "AB" & ASCII.DOLLAR & "CD". Similarly, the string literal "ABcd" with lower case letters could be replaced by "AB" & ASCII.LC_C & ASCII.LC_D.
- 8 *References:* ascii predefined package C, based literal 2.4.2, basic character 2.1, catenation operation 4.5.3, character value 3.5.2, delimiter 2.2, graphic character 2.1, graphical symbol 2.1, identifier 2.3, lexical element 2.2, lower case letter 2.1, numeric literal 2.4, string bracket 2.6, string literal 2.6, upper case letter 2.1

3. Declarations and Types

This chapter describes the types in the language and the rules for declaring constants, variables, and named numbers.

3.1 Declarations

The language defines several kinds of entities that are declared, either explicitly or implicitly, by declarations. Such an entity can be a numeric literal, an object, a discriminant, a record component, a loop parameter, an exception, a type, a subtype, a subprogram, a package, a task unit, a generic unit, a single entry, an entry family, a formal parameter (of a subprogram, entry, or generic subprogram), a generic formal parameter, a named block or loop, a labeled statement, or an operation (in particular, an attribute or an enumeration literal; see 3.3.3).

There are several forms of declaration. A basic declaration is a form of declaration defined as follows.

```
basic_declaration ::=
    object_declaration      | number_declaration
    | type_declaration       | subtype_declaration
    | subprogram_declaration | package_declaration
    | task_declaration       | generic_declaration
    | exception_declaration  | generic_instantiation
    | renaming_declaration   | deferred_constant_declaration
```

Certain forms of declaration always occur (explicitly) as part of a basic declaration; these forms are discriminant specifications, component declarations, entry declarations, parameter specifications, generic parameter declarations, and enumeration literal specifications. A loop parameter specification is a form of declaration that occurs only in certain forms of loop statement.

The remaining forms of declaration are implicit: the name of a block, the name of a loop, and a statement label are implicitly declared. Certain operations are implicitly declared (see 3.3.3).

For each form of declaration the language rules define a certain region of text called the *scope* of the declaration (see 8.2). Several forms of declaration 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 associated declared 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 (its simple name); the name is said to *denote* the associated entity.

Certain forms of enumeration literal specification associate a character literal with the corresponding declared entity. Certain forms of declaration associate an operator symbol or some other notation with an explicitly or implicitly declared operation.

The process by which a declaration achieves its effect is called the *elaboration* of the declaration; this process happens during program execution.

After its elaboration, a declaration is said to be *elaborated*. Prior to the completion of its elaboration (including before the elaboration), the declaration is not yet elaborated. The elaboration of any declaration has always at least the effect of achieving this change of state (from not yet elaborated to elaborated). The phrase “*the elaboration has no other effect*” is used in this manual whenever this change of state is the only effect of elaboration for some form of declaration. An elaboration process is also defined for declarative parts, declarative items, and compilation units (see 3.9 and 10.5).

Object, number, type, and subtype declarations are described here. The remaining basic declarations are described in later chapters.

Note:

The syntax rules use the term *identifier* for the first occurrence of an identifier in some form of declaration; the term *simple name* is used for any occurrence of an identifier that already denotes some declared entity.

References: attribute 4.1.4, block name 5.6, block statement 5.6, character literal 2.5, component declaration 3.7, declarative item 3.9, declarative part 3.9, deferred constant declaration 7.4, discriminant specification 3.7.1, elaboration 3.9, entry declaration 9.5, enumeration literal specification 3.5.1, exception declaration 11.1, generic declaration 12.1, generic instantiation 12.3, generic parameter declaration 12.1, identifier 2.3, label 5.1, loop name 5.5, loop parameter specification 5.5, loop statement 5.5, name 4.1, number declaration 3.2.2, numeric literal 2.4, object declaration 3.2.1, operation 3.3, operator symbol 6.1, package declaration 7.1, parameter specification 6.1, record component 3.7, renaming declaration 8.5, representation clause 13.1, scope 8.2, simple name 4.1, subprogram body 6.3, subprogram declaration 6.1, subtype declaration 3.3.2, task declaration 9.1, type declaration 3.3.1, visibility 8.3

3.2 Objects and Named Numbers

An *object* is an entity that contains (has) a value of a given type. An object is one of the following:

- an object declared by an object declaration or by a single task declaration,
- a formal parameter of a subprogram, entry, or generic subprogram,
- a generic formal object,
- a loop parameter,
- an object designated by a value of an access type,
- a component or a slice of another object.

A number declaration is a special form of object declaration that associates an identifier with a value of type *universal_integer* or *universal_real*.

```
object_declaration ::=
    identifier_list : [constant] subtype_indication [:= expression];
    | identifier_list : [constant] constrained_array_definition [:= expression];

number_declaration ::=
    identifier_list : constant := universal_static_expression;

identifier_list ::= identifier {, identifier}
```

An object declaration is called a *single object declaration* if its identifier list has a single identifier; it is called a *multiple object declaration* if the identifier list has two or more identifiers. A multiple object declaration is equivalent to a sequence of the corresponding number of single object declarations. For each identifier of the list, the equivalent sequence has a single object declaration formed by this identifier, followed by a colon and by whatever appears at the right of the colon in the multiple object declaration; the equivalent sequence is in the same order as the identifier list. 10

A similar equivalence applies also for the identifier lists of number declarations, component declarations, discriminant specifications, parameter specifications, generic parameter declarations, exception declarations, and deferred constant declarations. 11

In the remainder of this reference manual, explanations are given for declarations with a single identifier; the corresponding explanations for declarations with several identifiers follow from the equivalence stated above. 12

Example: 13

-- the multiple object declaration

JOHN, PAUL : PERSON_NAME := new PERSON(SEX => M); -- see 3.8.1

-- is equivalent to the two single object declarations in the order given

JOHN : PERSON_NAME := new PERSON(SEX => M);

PAUL : PERSON_NAME := new PERSON(SEX => M);

References: access type 3.8, constrained array definition 3.6, component 3.3, declaration 3.1, deferred constant declaration 7.4, designate 3.8, discriminant specification 3.7.1, entry 9.5, exception declaration 11.1, expression 4.4, formal parameter 6.1, generic formal object 12.1.1, generic parameter declaration 12.1, generic unit 12, generic subprogram 12.1, identifier 2.3, loop parameter 5.5, numeric type 3.5, parameter specification 6.1, scope 8.2, simple name 4.1, single task declaration 9.1, slice 4.1.2, static expression 4.9, subprogram 6, subtype indication 3.3.2, type 3.3, universal_integer type 3.5.4, universal_real type 3.5.6 14

3.2.1 Object Declarations

An object declaration declares an object whose type is given either by a subtype indication or by a constrained array definition. If the object declaration includes the assignment compound delimiter followed by an expression, the expression specifies an initial value for the declared object; the type of the expression must be that of the object. 1

The declared object is a *constant* if the reserved word **constant** appears in the object declaration; the declaration must then include an explicit initialization. The value of a constant cannot be modified after initialization. Formal parameters of mode **in** of subprograms and entries, and generic formal parameters of mode **in**, are also constants; a loop parameter is a constant within the corresponding loop; a subcomponent or slice of a constant is a constant. 2

An object that is not a constant is called a *variable* (in particular, the object declared by an object declaration that does not include the reserved word **constant** is a variable). The only ways to change the value of a variable are either directly by an assignment, or indirectly when the variable is updated (see 6.2) by a procedure or entry call statement (this action can be performed either on the variable itself, on a subcomponent of the variable, or on another variable that has the given variable as subcomponent). 3

4 The elaboration of an object declaration proceeds as follows:

- 5 (a) The subtype indication or the constrained array definition is first elaborated. This establishes the subtype of the object.
- 6 (b) If the object declaration includes an explicit initialization, the initial value is obtained by evaluating the corresponding expression. Otherwise any implicit initial values for the object or for its subcomponents are evaluated.
- 7 (c) The object is created.
- 8 (d) Any initial value (whether explicit or implicit) is assigned to the object or to the corresponding subcomponent.

9 Implicit initial values are defined for objects declared by object declarations, and for components of such objects, in the following cases:

- 10 • If the type of an object is an access type, the implicit initial value is the null value of the access type.
- 11 • If the type of an object is a task type, the implicit initial (and only) value designates a corresponding task.
- 12 • If the type of an object is a type with discriminants and the subtype of the object is constrained, the implicit initial (and only) value of each discriminant is defined by the subtype of the object.
- 13 • If the type of an object is a composite type, the implicit initial value of each component that has a default expression is obtained by evaluation of this expression, unless the component is a discriminant of a constrained object (the previous case).

14 In the case of a component that is itself a composite object and whose value is defined neither by an explicit initialization nor by a default expression, any implicit initial values for components of the composite object are defined by the same rules as for a declared object.

15 The steps (a) to (d) are performed in the order indicated. For step (b), if the default expression for a discriminant is evaluated, then this evaluation is performed before that of default expressions for subcomponents that depend on discriminants, and also before that of default expressions that include the name of the discriminant. Apart from the previous rule, the evaluation of default expressions is performed in some order that is not defined by the language.

16 The initialization of an object (the declared object or one of its subcomponents) checks that the initial value belongs to the subtype of the object; for an array object declared by an object declaration, an implicit subtype conversion is first applied as for an assignment statement, unless the object is a constant whose subtype is an unconstrained array type. The exception `CONSTRAINT_ERROR` is raised if this check fails.

17 The value of a scalar variable is undefined after elaboration of the corresponding object declaration unless an initial value is assigned to the variable by an initialization (explicitly or implicitly).

18 If the operand of a type conversion or qualified expression is a variable that has scalar subcomponents with undefined values, then the values of the corresponding subcomponents of the result are undefined. The execution of a program is erroneous if it attempts to evaluate a scalar variable with an undefined value. Similarly, the execution of a program is erroneous if it attempts to apply a predefined operator to a variable that has a scalar subcomponent with an undefined value.

3.2.1 Object Declarations

3-4

Examples of variable declarations:

19

```

COUNT, SUM    : INTEGER;
SIZE           : INTEGER range 0 .. 10_000 := 0;
SORTED         : BOOLEAN := FALSE;
COLOR_TABLE    : array(1 .. N) of COLOR;
OPTION         : BIT_VECTOR(1 .. 10) := (others => TRUE);

```

Examples of constant declarations:

20

```

LIMIT          : constant INTEGER := 10_000;
LOW_LIMIT      : constant INTEGER := LIMIT/10;
TOLERANCE      : constant REAL := DISPERSION(1.15);

```

Note:

The expression initializing a constant object need not be a static expression (see 4.9). In the above examples, LIMIT and LOW_LIMIT are initialized with static expressions, but TOLERANCE is not if DISPERSION is a user-defined function.

21

References: access type 3.8, assignment 5.2, assignment compound delimiter 5.2, component 3.3, composite type 3.3, constrained array definition 3.6, constrained subtype 3.3, constraint_error exception 11.1, conversion 4.6, declaration 3.1, default expression for a discriminant 3.7, default initial value for an access type 3.8, depend on a discriminant 3.7.1, designate 3.8, discriminant 3.3, elaboration 3.9, entry 9.5, evaluation 4.5, expression 4.4, formal parameter 6.1, generic formal parameter 12.1 12.3, generic unit 12, in some order 1.6, limited type 7.4.4, mode in 6.1, package 7, predefined operator 4.5, primary 4.4, private type 7.4, qualified expression 4.7, reserved word 2.9, scalar type 3.5, slice 4.1.2, subcomponent 3.3, subprogram 6, subtype 3.3, subtype indication 3.3.2, task 9, task type 9.2, type 3.3, visible part 7.2

22

3.2.2 Number Declarations

A number declaration is a special form of constant declaration. The type of the static expression given for the initialization of a number declaration must be either the type *universal_integer* or the type *universal_real*. The constant declared by a number declaration is called a *named number* and has the type of the static expression.

1

Note:

The rules concerning expressions of a universal type are explained in section 4.10. It is a consequence of these rules that if every primary contained in the expression is of the type *universal_integer*, then the named number is also of this type. Similarly, if every primary is of the type *universal_real*, then the named number is also of this type.

2

Examples of number declarations:

3

```

PI              : constant := 3.14159_26536; -- a real number
TWO_PI          : constant := 2.0*PI;        -- a real number
MAX             : constant := 500;            -- an integer number
POWER_16        : constant := 2**16;         -- the integer 65_536
ONE, UN, EINS   : constant := 1;             -- three different names for 1

```

References: identifier 2.3, primary 4.4, static expression 4.9, type 3.3, universal_integer type 3.5.4, universal_real type 3.5.6, universal type 4.10

4

3.3 Types and Subtypes

- 1 A type is characterized by a set of values and a set of operations.
 - 2 There exist several *classes* of types. *Scalar* types are integer types, real types, and types defined by enumeration of their values; values of these types have no components. *Array* and *record* types are composite; a value of a composite type consists of *component* values. An *access* type is a type whose values provide access to objects. *Private* types are types for which the set of possible values is well defined, but not directly available to the users of such types. Finally, there are *task* types. (Private types are described in chapter 7, task types are described in chapter 9, the other classes of types are described in this chapter.)
 - 3 Certain record and private types have special components called *discriminants* whose values distinguish alternative forms of values of one of these types. If a private type has discriminants, they are known to users of the type. Hence a private type is only known by its name, its discriminants if any, and by the corresponding set of operations.
 - 4 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 where the constraint imposes no restriction is also included); a value is said to *satisfy* a constraint if it satisfies the corresponding condition. A *subtype* is a type together with a constraint; a value is said to *belong to a subtype* of a given type if it belongs to the type and satisfies the constraint; the given type is called the *base type* of the subtype. A type is a subtype of itself; such a subtype is said to be *unconstrained*: it corresponds to a condition that imposes no restriction. The base type of a type is the type itself.
 - 5 The set of operations defined for a subtype of a given type includes the operations that are defined for the type; however the assignment operation to a variable having a given subtype only assigns values that belong to the subtype. Additional operations, such as qualification (in a qualified expression), are implicitly defined by a subtype declaration.
 - 6 Certain types have *default initial values* defined for objects of the type; certain other types have *default expressions* defined for some or all of their components. Certain operations of types and subtypes are called *attributes*; these operations are denoted by the form of name described in section 4.1.4.
 - 7 The term *subcomponent* is used in this manual in place of the term component to indicate either a component, or a component of another component or subcomponent. Where other subcomponents are excluded, the term component is used instead.
 - 8 A given type must not have a subcomponent whose type is the given type itself.
 - 9 The name of a class of types is used in this manual as a qualifier for objects and values that have a type of the class considered. For example, the term "array object" is used for an object whose type is an array type; similarly, the term "access value" is used for a value of an access type.
- Note:*
- 10 The set of values of a subtype is a subset of the values of the base type. This subset need not be a proper subset; it can be an empty subset.
 - 11 *References:* access type 3.8, array type 3.6, assignment 5.2, attribute 4.1.4, component of an array 3.6, component of a record 3.7, discriminant constraint 3.7.2, enumeration type 3.5.1, integer type 3.5.4, object 3.2.1, private type 7.4, qualified expression 4.7, real type 3.5.6, record type 3.7, subtype declaration 3.3.2, task type 9.1, type declaration 3.3.1

3.3.1 Type Declarations

A type declaration declares a type.

```

type_declaration ::= full_type_declaration
                  | incomplete_type_declaration | private_type_declaration

full_type_declaration ::=
    type identifier [discriminant_part] is type_definition;

type_definition ::=
    enumeration_type_definition | integer_type_definition
    | real_type_definition      | array_type_definition
    | record_type_definition    | access_type_definition
    | derived_type_definition

```

The elaboration of a full type declaration consists of the elaboration of the discriminant part, if any (except in the case of the full type declaration for an incomplete or private type declaration), and of the elaboration of the type definition.

The types created by the elaboration of distinct type definitions are distinct types. Moreover, the elaboration of the type definition for a numeric or derived type creates both a base type and a subtype of the base type; the same holds for a constrained array definition (one of the two forms of array type definition).

The simple name declared by a full type declaration denotes the declared type, unless the type declaration declares both a base type and a subtype of the base type, in which case the simple name denotes the subtype, and the base type is anonymous. A type is said to be *anonymous* if it has no simple name. For explanatory purposes, this reference manual 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.

Examples of type definitions:

```

(WHITE, RED, YELLOW, GREEN, BLUE, BROWN, BLACK)
range 1 .. 72
array(1 .. 10) of INTEGER

```

Examples of type declarations:

```

type COLOR    is (WHITE, RED, YELLOW, GREEN, BLUE, BROWN, BLACK);
type COLUMN   is range 1 .. 72;
type TABLE   is array(1 .. 10) of INTEGER;

```

Notes:

Two type definitions always define two distinct types, even if they are textually identical. Thus, the array type definitions given in the declarations of A and B below define distinct types.

```

A : array(1 .. 10) of BOOLEAN;
B : array(1 .. 10) of BOOLEAN;

```

If A and B are declared by a multiple object declaration as below, their types are nevertheless different, since the multiple object declaration is equivalent to the above two single object declarations.

```

A, B : array(1 .. 10) of BOOLEAN;

```

Incomplete type declarations are used for the definition of recursive and mutually dependent types (see 3.8.1). Private type declarations are used in package specifications and in generic parameter declarations (see 7.4 and 12.1).

References: access type definition 3.8, array type definition 3.6, base type 3.3, constrained array definition 3.6, constrained subtype 3.3, declaration 3.1, derived type 3.4, derived type definition 3.4, discriminant part 3.7.1, elaboration 3.9, enumeration type definition 3.5.1, identifier 2.3, incomplete type declaration 3.8.1, integer type definition 3.5.4, multiple object declaration 3.2, numeric type 3.5, private type declaration 7.4, real type definition 3.5.6, reserved word 2.9, type 3.3

3.3.2 Subtype Declarations

A subtype declaration declares a subtype.

```

subtype_declaration ::=
    subtype identifier is subtype_indication;

subtype_indication ::= type_mark [constraint]

type_mark ::= type_name | subtype_name

constraint ::=
    range_constraint | floating_point_constraint | fixed_point_constraint
    | index_constraint | discriminant_constraint

```

A type mark denotes a type or a subtype. If a type mark is the name of a type, the type mark denotes this type and also the corresponding unconstrained subtype. The *base type of a type mark* is, by definition, the base type of the type or subtype denoted by the type mark.

A subtype indication defines a subtype of the base type of the type mark.

If an index constraint appears after a type mark in a subtype indication, the type mark must not already impose an index constraint. Likewise for a discriminant constraint, the type mark must not already impose a discriminant constraint.

The elaboration of a subtype declaration consists of the elaboration of the subtype indication. The elaboration of a subtype indication creates a subtype. If the subtype indication does not include a constraint, the subtype is the same as that denoted by the type mark. The elaboration of a subtype indication that includes a constraint proceeds as follows:

(a) The constraint is first elaborated.

(b) A check is then made that the constraint is *compatible* with the type or subtype denoted by the type mark.

The condition imposed by a constraint is the condition obtained after elaboration of the constraint. (The rules of constraint elaboration are such that the expressions and ranges of constraints are evaluated by the elaboration of these constraints.) The rules defining compatibility are given for each form of constraint in the appropriate section. 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.

Examples of subtype declarations:

```

subtype RAINBOW is COLOR range RED .. BLUE;           -- see 3.3.1
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 3.3.1
subtype SQUARE is MATRIX(1 .. 10, 1 .. 10);             -- see 3.6
subtype MALE is PERSON(SEX => M);                         -- see 3.8

```

Note:

A subtype declaration does not define a new type.

References: base type 3.3, compatibility of discriminant constraints 3.7.2, compatibility of fixed point constraints 3.5.9, compatibility of floating point constraints 3.5.7, compatibility of index constraints 3.6.1, compatibility of range constraints 3.5, constraint_error exception 11.1, declaration 3.1, discriminant 3.3, discriminant constraint 3.7.2, elaboration 3.9, evaluation 4.5, expression 4.4, floating point constraint 3.5.7, fixed point constraint 3.5.9, index constraint 3.6.1, range constraint 3.5, reserved word 2.9, subtype 3.3, type 3.3, type name 3.3.1, unconstrained subtype 3.3

3.3.3 Classification of Operations

The set of operations of a type includes the explicitly declared subprograms that have a parameter or result of the type; such subprograms are necessarily declared after the type declaration.

The remaining operations are each implicitly declared for a given type declaration, immediately after the type definition. These implicitly declared operations comprise the *basic* operations, the predefined operators (see 4.5), and enumeration literals. In the case of a derived type declaration, the implicitly declared operations include any derived subprograms. The operations implicitly declared for a given type declaration occur after the type declaration and before the next explicit declaration, if any. The implicit declarations of derived subprograms occur last.

A basic operation is an operation that is inherent in one of the following:

- An assignment (in assignment statements and initializations), an allocator, a membership test, or a short-circuit control form.
- A selected component, an indexed component, or a slice.
- A qualification (in qualified expressions), an explicit type conversion, or an implicit type conversion of a value of type *universal_integer* or *universal_real* to the corresponding value of another numeric type.
- A numeric literal (for a universal type), the literal **null** (for an access type), a string literal, an aggregate, or an attribute.

For every type or subtype T, the following attribute is defined:

T'BASE The base type of T. This attribute is allowed only as the prefix of the name of another attribute: for example, T'BASE'FIRST.

Note:

Each literal is an operation whose evaluation yields the corresponding value (see 4.2). Likewise, an aggregate is an operation whose evaluation yields a value of a composite type (see 4.3). Some operations of a type *operate on* values of the type, for example, predefined operators and certain subprograms and attributes. The evaluation of some operations of a type *returns* a value of the type, for example, literals and certain functions, attributes, and predefined operators. Assignment is an operation that operates on an object and a value. The evaluation of the operation corresponding to a selected component, an indexed component, or a slice, yields the object or value denoted by this form of name.

References: aggregate 4.3, allocator 4.8, assignment 5.2, attribute 4.1.4, character literal 2.5, composite type 3.3, conversion 4.6, derived subprogram 3.4, enumeration literal 3.5.1, formal parameter 6.1, function 6.5, indexed component 4.1.1, initial value 3.2.1, literal 4.2, membership test 4.5 4.5.2, null literal 3.8, numeric literal 2.4, numeric type 3.5, object 3.2.1, 6.1, predefined operator 4.5, qualified expression 4.7, selected component 4.1.3, short-circuit control form 4.5 4.5.1, slice 4.1.2, string literal 2.6, subprogram 6, subtype 3.3, type 3.3, type declaration 3.3.1, universal_integer type 3.5.4, universal_real type 3.5.6, universal type 4.10

3.4 Derived Types

A derived type definition defines a new (base) type whose characteristics are derived from those of a *parent type*; the new type is called a *derived type*. A derived type definition further defines a *derived subtype*, which is a subtype of the derived type.

derived_type_definition ::= **new** subtype_indication

The subtype indication that occurs after the reserved word **new** defines the *parent subtype*. The parent type is the base type of the parent subtype. If a constraint exists for the parent subtype, a similar constraint exists for the derived subtype; the only difference is that for a range constraint, and likewise for a floating or fixed point constraint that includes a range constraint, the value of each bound is replaced by the corresponding value of the derived type. The characteristics of the derived type are defined as follows:

- The derived type belongs to the same class of types as the parent type. The set of possible values for the derived type is a copy of the set of possible values for the parent type. If the parent type is composite, then the same components exist for the derived type, and the subtype of corresponding components is the same.
- For each basic operation of the parent type, there is a corresponding basic operation of the derived type. Explicit type conversion of a value of the parent type into the corresponding value of the derived type is allowed and vice versa as explained in section 4.6.
- For each enumeration literal or predefined operator of the parent type there is a corresponding operation for the derived type.
- If the parent type is a task type, then for each entry of the parent type there is a corresponding entry for the derived type.
- If a default expression exists for a component of an object having the parent type, then the same default expression is used for the corresponding component of an object having the derived type.

- If the parent type is an access type, then the parent and the derived type share the same collection; there is a null access value for the derived type and it is the default initial value of that type. 9
- If an explicit representation clause exists for the parent type and if this clause appears before the derived type definition, then there is a corresponding representation clause (an implicit one) for the derived type. 10
- Certain subprograms that are operations of the parent type are said to be *derivable*. For each derivable subprogram of the parent type, there is a corresponding derived subprogram for the derived type. Two kinds of derivable subprograms exist. First, if the parent type is declared immediately within the visible part of a package, then a subprogram that is itself explicitly declared immediately within the visible part becomes derivable after the end of the visible part, if it is an operation of the parent type. (The explicit declaration is by a subprogram declaration, a renaming declaration, or a generic instantiation.) Second, if the parent type is itself a derived type, then any subprogram that has been derived by this parent type is further derivable, unless the parent type is declared in the visible part of a package and the derived subprogram is hidden by a derivable subprogram of the first kind. 11

Each operation of the derived type is implicitly declared at the place of the derived type declaration. The implicit declarations of any derived subprograms occur last. 12

The specification of a derived subprogram is obtained implicitly by systematic replacement of the parent type by the derived type in the specification of the derivable subprogram. Any subtype of the parent type is likewise replaced by a subtype of the derived type with a similar constraint (as for the transformation of a constraint of the parent subtype into the corresponding constraint of the derived subtype). Finally, any expression of the parent type is made to be the operand of a type conversion that yields a result of the derived type. 13

Calling a derived subprogram is equivalent to calling the corresponding subprogram of the parent type, in which each actual parameter that is of the derived type is replaced by a type conversion of this actual parameter to the parent type (this means that a conversion to the parent type happens before the call for the modes **in** and **in out**; a reverse conversion to the derived type happens after the call for the modes **in out** and **out**, see 6.4.1). In addition, if the result of a called function is of the parent type, this result is converted to the derived type. 14

If a derived or private type is declared immediately within the visible part of a package, then, within this visible part, this type must not be used as the parent type of a derived type definition. (For private types, see also section 7.4.1.) 15

For the elaboration of a derived type definition, the subtype indication is first elaborated, the derived type is then created, and finally, the derived subtype is created. 16

Examples: 17

```

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

type SPECIAL_KEY is new KEY_MANAGER.KEY;    -- see 7.4.2
-- the derived subprograms have the following specifications:

-- procedure GET_KEY(K : out SPECIAL_KEY);
-- function "<"(X,Y : SPECIAL_KEY) return BOOLEAN;
```

Notes:

- 18 The rules of derivation of basic operations and enumeration literals imply that the notation for any literal or aggregate of the derived type is the same as for the parent type; such literals and aggregates are said to be *overloaded*. Similarly, it follows that the notation for denoting a component, a discriminant, an entry, a slice, or an attribute is the same for the derived type as for the parent type.
- 19 Hiding of a derived subprogram is allowed even within the same declarative region (see 8.3). A derived subprogram hides a predefined operator that has the same parameter and result type profile (see 6.6).
- 20 A generic subprogram declaration is not derivable since it declares a generic unit rather than a subprogram. On the other hand, an instantiation of a generic subprogram is a (nongeneric) subprogram, which is derivable if it satisfies the requirements for derivability of subprograms.
- 21 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).
- 22 If a representation clause is given for the parent type but appears after the derived type declaration, then no corresponding representation clause applies to the derived type; hence an explicit representation clause for such a derived type is allowed.
- 23 For a derived subprogram, if a parameter belongs to the derived type, the subtype of this parameter need not have any value in common with the derived subtype.
- 24 *References:* access value 3.8, actual parameter 6.4.1, aggregate 4.3, attribute 4.1.4, base type 3.3, basic operation 3.3.3, boolean type 3.5.3, bound of a range 3.5, class of type 3.3, collection 3.8, component 3.3, composite type 3.3, constraint 3.3, conversion 4.6, declaration 3.1, declarative region 8.1, default expression 3.2.1, default initial value for an access type 3.8, discriminant 3.3, elaboration 3.9, entry 9.5, enumeration literal 3.5.1, floating point constraint 3.5.7, fixed point constraint 3.5.9, formal parameter 6.1, function call 6.4, generic declaration 12.1, immediately within 8.1, implicit declaration 3.1, literal 4.2, mode 6.1, overloading 6.6 8.7, package 7, package specification 7.1, parameter association 6.4, predefined operator 4.5, private type 7.4, procedure 6, procedure call statement 6.4, range constraint 3.5, representation clause 13.1, reserved word 2.9, slice 4.1.2, subprogram 6, subprogram specification 6.1, subtype indication 3.3.2, subtype 3.3, type 3.3, type definition 3.3.1, visible part 7.2

3.5 Scalar Types

- 1 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.

- 2 range_constraint ::= **range** range
- range ::= *range_attribute*
 | simple_expression .. simple_expression

A range specifies a subset of values of a scalar type. The range $L .. R$ specifies the values from L to R inclusive if the relation $L \leq R$ is true. The values L and R are called the *lower bound* and *upper bound* of the range, respectively. A value V is said to *satisfy* a range constraint if it belongs to the range; the value V is said to *belong* to the range if the relations $L \leq V$ and $V \leq R$ are both TRUE. A *null* range is a range for which the relation $R < L$ is TRUE; no value belongs to a null range. The operators $\leq$ and $<$ in the above definitions are the predefined operators of the scalar type.

If a range constraint is used in a subtype indication, either directly or as part of a floating or fixed point constraint, the type of the simple expressions (likewise, of the bounds of a range attribute) must be the same as the base type of the type mark of the subtype indication. A range constraint is *compatible* with a subtype if each bound of the range belongs to the subtype, or if the range constraint defines a null range; otherwise the range constraint is not compatible with the subtype.

The elaboration of a range constraint consists of the evaluation of the range. The evaluation of a range defines its lower bound and its upper bound. If simple expressions are given to specify the bounds, the evaluation of the range evaluates these simple expressions in some order that is not defined by the language.

Attributes

For any scalar type T or for any subtype T of a scalar type, the following attributes are defined:

T .FIRST Yields the lower bound of T . The value of this attribute has the same type as T .

T .LAST Yields the upper bound of T . The value of this attribute has the same type as T .

Note:

Indexing and iteration rules use values of discrete types.

References: attribute 4.1.4, constraint 3.3, enumeration type 3.5.1, erroneous 1.6, evaluation 4.5, fixed point constraint 3.5.9, floating point constraint 3.5.7, index 3.6, integer type 3.5.4, loop statement 5.5, range attribute 3.6.2, real type 3.5.6, relational operator 4.5 4.5.2, satisfy a constraint 3.3, simple expression 4.4, subtype indication 3.3.2, type mark 3.3.2

3.5.1 Enumeration Types

An enumeration type definition defines an enumeration type.

```
enumeration_type_definition ::=
    (enumeration_literal_specification {, enumeration_literal_specification})

enumeration_literal_specification ::= enumeration_literal

enumeration_literal ::= identifier | character_literal
```

The identifiers and character literals listed by an enumeration type definition must be distinct. Each enumeration literal specification is the declaration of the corresponding enumeration literal: this declaration is equivalent to the declaration of a parameterless function, the designator being the enumeration literal, and the result type being the enumeration type. The elaboration of an enumeration type definition creates an enumeration type; this elaboration includes that of every enumeration literal specification.

Each enumeration literal yields a different enumeration value. The predefined order relations between enumeration values follow the order of corresponding position numbers. The position number of the value of the first listed enumeration literal is zero; the position number for each other enumeration literal is one more than for its predecessor in the list.

If the same identifier or character literal is specified in more than one enumeration type definition, the corresponding 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 must be determinable from the context (see 8.7).

Examples:

```

type DAY      is (MON, TUE, WED, THU, FRI, SAT, SUN);
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');
type MIXED    is ('A', 'B', '*', B, NONE, '?', '%');

subtype WEEKDAY is DAY      range MON .. FRI;
subtype MAJOR   is SUIT     range HEARTS .. SPADES;
subtype RAINBOW is COLOR   range RED .. BLUE; -- the color RED, not the light

```

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 8.7).

References: character literal 2.5, declaration 3.1, designator 6.1, elaboration 3.9, 6.1, function 6.5, identifier 2.3, name 4.1, overloading 6.6 8.7, position number 3.5, qualified expression 4.7, relational operator 4.5 4.5.2, type 3.3, type definition 3.3.1

3.5.2 Character Types

An enumeration type is said to be a character type if at least one of its enumeration literals is a character literal. The predefined type CHARACTER is a character type whose values are the 128 characters of the *ASCII* character set. Each of the 95 graphic characters of this character set is denoted by the corresponding character literal.

Example:

```

type ROMAN_DIGIT is ('I', 'V', 'X', 'L', 'C', 'D', 'M');

```

Notes:

The predefined package ASCII includes the declaration of constants denoting control characters and of constants denoting graphic characters that are not in the basic character set.

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 section 13.3.

References: *ascii* predefined package C, basic character 2.1, character literal 2.5, constant 3.2.1, declaration 3.1, enumeration type 3.5.1, graphic character 2.1, identifier 2.3, literal 4.2, predefined type C, type 3.3

3.5.3 Boolean Types

There is a predefined enumeration type named `BOOLEAN`. It contains the two literals `FALSE` and `TRUE` ordered with the relation `FALSE < TRUE`. A boolean type is either the type `BOOLEAN` or a type that is derived, directly or indirectly, from a boolean type.

References: derived type 3.4, enumeration literal 3.5.1, enumeration type 3.5.1, relational operator 4.5 4.5.2, type 3.3

3.5.4 Integer Types

An integer type definition defines an integer type whose set of values includes at least those of the specified range.

```
integer_type_definition ::= range_constraint
```

If a range constraint is used as an integer type definition, each bound of the range must be defined by a static expression of some integer type, but the two bounds need not have the same integer type. (Negative bounds are allowed.)

A type declaration of the form:

```
type T is range L .. R;
```

is, by definition, equivalent to the following declarations:

```
type integer_type is new predefined_integer_type;
subtype T is integer_type range integer_type(L) .. integer_type(R);
```

where *integer_type* is an anonymous type, and where the predefined integer type is implicitly selected by the implementation, so as to contain the values *L* to *R* inclusive. The integer type declaration is illegal if none of the predefined integer types satisfies this requirement, excepting *universal_integer*. The elaboration of the declaration of an integer type consists of the elaboration of the equivalent type and subtype declarations.

The predefined integer types include the type `INTEGER`. An implementation may also have predefined types such as `SHORT_INTEGER` and `LONG_INTEGER`, which have (substantially) shorter and longer ranges, respectively, than `INTEGER`. The range of each of these types must be symmetric about zero, excepting an extra negative value which may exist in some implementations. The base type of each of these types is the type itself.

Integer literals are the literals of an anonymous predefined integer type that is called *universal_integer* in this reference manual. Other integer types have no literals. However, for each integer type there exists an implicit conversion that converts a *universal_integer* value into the corresponding value (if any) of the integer type. The circumstances under which these implicit conversions are invoked are described in section 4.6.

The position number of an integer value is the corresponding value of the type *universal_integer*.

The same arithmetic operators are predefined for all integer types (see 4.5). The exception `NUMERIC_ERROR` is raised by the execution of an operation (in particular an implicit conversion) that cannot deliver the correct result (that is, if the value corresponding to the mathematical result is not a value of the integer type). However, an implementation is not required to raise the exception `NUMERIC_ERROR` if the operation is part of a larger expression whose result can be computed correctly, as described in section 11.6.

Examples:

```
type PAGE_NUM is range 1 .. 2_000;
type LINE_SIZE is range 1 .. MAX_LINE_SIZE;

subtype SMALL_INT is INTEGER range -10 .. 10;
subtype COLUMN_PTR is LINE_SIZE range 1 .. 10;
subtype BUFFER_SIZE is INTEGER range 0 .. MAX;
```

Notes:

The name declared by an integer type declaration is a subtype name. On the other hand, the predefined operators of an integer type deliver results whose range is defined by the parent predefined type; such a result need not belong to the declared subtype, in which case an attempt to assign the result to a variable of the integer subtype raises the exception `CONSTRAINT_ERROR`.

The smallest (most negative) value supported by the predefined integer types of an implementation is the named number `SYSTEM.MIN_INT` and the largest (most positive) value is `SYSTEM.MAX_INT` (see 13.7).

References: anonymous type 3.3.1, belong to a subtype 3.3, bound of a range 3.5, `constraint_error` exception 11.1, conversion 4.6, identifier 2.3, integer literal 2.4, literal 4.2, `numeric_error` exception 11.1, parent type 3.4, predefined operator 4.5, range constraint 3.5, static expression 4.9, subtype declaration 3.3.2, system predefined package 13.7, type 3.3, type declaration 3.3.1, type definition 3.3.1, universal type 4.10

3.5.5 Operations of Discrete Types

The basic operations of a discrete type include the operations involved in assignment, the membership tests, and qualification; for a boolean type they include the short-circuit control forms; for an integer type they include the explicit conversion of values of other numeric types to the integer type, and the implicit conversion of values of the type *universal_integer* to the type.

Finally, for every discrete type or subtype *T*, the basic operations include the attributes listed below. In this presentation, *T* is referred to as being a subtype (the subtype *T*) for any property that depends on constraints imposed by *T*; other properties are stated in terms of the base type of *T*.