Lecture Notes in Computer Science

Edited by G. Goos and J. Hartmanis

106

The Programming Language Ada

Reference Manual

Proposed Standard Document
United States Department of Defense

Springer-Verlag
Berlin Heidelberg New York

Lecture Notes in Computer Science

Edited by G. Goos and J. Hartmanis

106

The Programming Language Ada

Reference Manual

Proposed Standard Document
United States Department of Defense

Springer-Verlag
Berlin Heidelberg New York 1981

Honeywell, Inc.
Systems and Research Center
2600 Ridgway Parkway
Minneapolis, MN 55413/USA
and
Cii Honeywell Bull
68 Route de Versailles
78430 Louveciennes, France

ISBN 3-540-10693-6 Springer-Verlag Berlin Heidelberg New York
ISBN 0-387-10693-6 Springer-Verlag New York Heidelberg Berlin

Printed in Germany

Printing and binding: Beltz Offsetdruck, Hemsbach/Bergstr.
2145/3140-54321

Lecture Notes in Computer Science

Vol. 1: GI-Gesellschaft für Informatik e.V. 3. Jahrestagung, Hamburg, 8.–10. Oktober 1973. Herausgegeben im Auftrag der Gesellschaft für Informatik von W. Brauer. XI, 508 Seiten. 1973.

Vol. 2: GI-Gesellschaft für Informatik e.V. 1. Fachtagung über Automatentheorie und Formale Sprachen, Bonn, 9.–12. Juli 1973. Herausgegeben im Auftrag der Gesellschaft für Informatik von K.-H. Böhling und K. Indermark. VII, 322 Seiten. 1973.

Vol. 3: 5th Conference on Optimization Techniques, Part I. (Series: I.F.I.P. TC7 Optimization Conferences.) Edited by R. Conti and A. Ruberti. XIII, 565 pages. 1973.

Vol. 4: 5th Conference on Optimization Techniques, Part II. (Series: I.F.I.P. TC7 Optimization Conferences.) Edited by R. Conti and A. Ruberti. XIII, 389 pages. 1973.

Vol. 5: International Symposium on Theoretical Programming. Edited by A. Ershov and V. A. Nepomniaschy. VI, 407 pages. 1974.

Vol. 6: B. T. Smith, J. M. Boyle, J. J. Dongarra, B. S. Garbow, Y. Ikebe, V. C. Klema, and C. B. Moler, Matrix Eigensystem Routines – EISPACK Guide. XI, 551 pages. 2nd Edition 1974. 1976.

Vol. 7: 3. Fachtagung über Programmiersprachen, Kiel, 5.–7. März 1974. Herausgegeben von B. Schlender und W. Frielinghaus. VI, 225 Seiten. 1974.

Vol. 8: GI-NTG Fachtagung über Struktur und Betrieb von Rechensystemen, Braunschweig, 20.–22. März 1974. Herausgegeben im Auftrag der GI und der NTG von H.-O. Leilich. VI, 340 Seiten. 1974.

Vol. 9: GI-BIFOA Internationale Fachtagung: Informationszentren in Wirtschaft und Verwaltung. Köln, 17./18. Sept. 1973. Herausgegeben im Auftrag der GI und dem BIFOA von P. Schmitz. VI, 259 Seiten. 1974.

Vol. 10: Computing Methods in Applied Sciences and Engineering, Part 1. International Symposium, Versailles, December 17–21, 1973. Edited by R. Glowinski and J. L. Lions. X, 497 pages. 1974.

Vol. 11: Computing Methods in Applied Sciences and Engineering, Part 2. International Symposium, Versailles, December 17–21, 1973. Edited by R. Glowinski and J. L. Lions. X, 434 pages. 1974.

Vol. 12: GFK-GI-GMR Fachtagung Prozessrechner 1974. Karlsruhe, 10.–11. Juni 1974. Herausgegeben von G. Krüger und R. Friehmelt. XI, 620 Seiten. 1974.

Vol. 13: Rechnerstrukturen und Betriebsprogrammierung, Erlangen, 1970. (GI-Gesellschaft für Informatik e.V.) Herausgegeben von W. Händler und P. P. Spies. VII, 333 Seiten. 1974.

Vol. 14: Automata, Languages and Programming – 2nd Colloquium, University of Saarbrücken, July 29–August 2, 1974. Edited by J. Loeckx. VIII, 611 pages. 1974.

Vol. 15: L Systems. Edited by A. Salomaa and G. Rozenberg. VI, 338 pages. 1974.

Vol. 16: Operating Systems, International Symposium, Rocquencourt 1974. Edited by E. Gelenbe and C. Kaiser. VIII, 310 pages. 1974.

Vol. 17: Rechner-Gestützter Unterricht RGU '74, Fachtagung, Hamburg, 12.–14. August 1974, ACU-Arbeitskreis Computer-Unterstützter Unterricht. Herausgegeben im Auftrag der GI von K. Brunnstein, K. Haefner und W. Händler. X, 417 Seiten. 1974.

Vol. 18: K. Jensen and N. E. Wirth, PASCAL – User Manual and Report. VII, 170 pages. Corrected Reprint of the 2nd Edition 1976.

Vol. 19: Programming Symposium. Proceedings 1974. V, 425 pages. 1974.

Vol. 20: J. Engelfriet, Simple Program Schemes and Formal Languages. VII, 254 pages. 1974.

Vol. 21: Compiler Construction, An Advanced Course. Edited by F. L. Bauer and J. Eickel. XIV. 621 pages. 1974.

Vol. 22: Formal Aspects of Cognitive Processes. Proceedings 1972. Edited by T. Storer and D. Winter. V, 214 pages. 1975.

Vol. 23: Programming Methodology. 4th Informatik Symposium, IBM Germany Wildbad, September 25–27, 1974. Edited by C. E. Hackl. VI, 501 pages. 1975.

Vol. 24: Parallel Processing. Proceedings 1974. Edited by T. Feng. VI, 433 pages. 1975.

Vol. 25: Category Theory Applied to Computation and Control. Proceedings 1974. Edited by E. G. Manes. X, 245 pages. 1975.

Vol. 26: GI-4. Jahrestagung, Berlin, 9.–12. Oktober 1974. Herausgegeben im Auftrag der GI von D. Siefkes. IX, 748 Seiten. 1975.

Vol. 27: Optimization Techniques. IFIP Technical Conference. Novosibirsk, July 1–7, 1974. (Series: I.F.I.P. TC7 Optimization Conferences.) Edited by G. I. Marchuk. VIII, 507 pages. 1975.

Vol. 28: Mathematical Foundations of Computer Science. 3rd Symposium at Jadwisin near Warsaw, June 17–22, 1974. Edited by A. Blikle. VII, 484 pages. 1975.

Vol. 29: Interval Mathematics. Procedings 1975. Edited by K. Nickel. VI, 331 pages. 1975.

Vol. 30: Software Engineering. An Advanced Course. Edited by F. L. Bauer. (Formerly published 1973 as Lecture Notes in Economics and Mathematical Systems, Vol. 81) XII, 545 pages. 1975.

Vol. 31: S. H. Fuller, Analysis of Drum and Disk Storage Units. IX, 283 pages. 1975.

Vol. 32: Mathematical Foundations of Computer Science 1975. Proceedings 1975. Edited by J. Bečvář. X, 476 pages. 1975.

Vol. 33: Automata Theory and Formal Languages, Kaiserslautern, May 20–23, 1975. Edited by H. Brakhage on behalf of GI. VIII, 292 Seiten. 1975.

Vol. 34: GI – 5. Jahrestagung, Dortmund 8.–10. Oktober 1975. Herausgegeben im Auftrag der GI von J. Mühlbacher. X, 755 Seiten. 1975.

Vol. 35: W. Everling, Exercises in Computer Systems Analysis. (Formerly published 1972 as Lecture Notes in Economics and Mathematical Systems, Vol. 65) VIII, 184 pages. 1975.

Vol. 36: S. A. Greibach, Theory of Program Structures: Schemes, Semantics, Verification. XV, 364 pages. 1975.

Vol. 37: C. Böhm, λ-Calculus and Computer Science Theory. Proceedings 1975. XII, 370 pages. 1975.

Vol. 38: P. Branquart, J.-P. Cardinael, J. Lewi, J.-P. Delescaille, M. Vanbegin. An Optimized Translation Process and Its Application to ALGOL 68. IX, 334 pages. 1976.

Vol. 39: Data Base Systems. Proceedings 1975. Edited by H. Hasselmeier and W. Spruth. VI, 386 pages. 1976.

Vol. 40: Optimization Techniques. Modeling and Optimization in the Service of Man. Part 1. Proceedings 1975. Edited by J. Cea. XIV, 854 pages. 1976.

Vol. 41: Optimization Techniques. Modeling and Optimization in the Service of Man. Part 2. Proceedings 1975. Edited by J. Cea. XIII, 852 pages. 1976.

Vol. 42: James E. Donahue, Complementary Definitions of Programming Language Semantics. VII, 172 pages. 1976.

Vol. 43: E. Specker und V. Strassen, Komplexität von Entscheidungsproblemen. Ein Seminar. V, 217 Seiten. 1976.

Vol. 44: ECI Conference 1976. Proceedings 1976. Edited by K. Samelson. VIII, 322 pages. 1976.

Vol. 45: Mathematical Foundations of Computer Science 1976. Proceedings 1976. Edited by A. Mazurkiewicz. XI, 601 pages. 1976.

Vol. 46: Language Hierarchies and Interfaces. Edited by F. L. Bauer and K. Samelson. X, 428 pages. 1976.

Vol. 47: Methods of Algorithmic Language Implementation. Edited by A. Ershov and C. H. A. Koster. VIII, 351 pages. 1977.

Vol. 48: Theoretical Computer Science, Darmstadt, March 1977. Edited by H. Tzschach, H. Waldschmidt and H. K.-G. Walter on behalf of GI. VIII, 418 pages. 1977.

EDITORS NOTE

This edition of the Ada Reference Manual is a photographic reproduction of the official November 1980 printing (Honeywell, Minneapolis). Because of the photo composition process, some errors were introduced in the November 1980 version which did not exist in the July 1980 version. These are listed below.

Section	Corrections
Table of contents	Change section numbers: "2–5" into "2.5" "2–6" into "2.6" "1–7" into "2.7"
03.05.05	In page 3–12, in T'SUCC(X), change ":item T'PRED(X) 11 The" into "T'PRED(X)" at the beginning of a new line and "The" tabulated as the previous lines.
03.07	Top of printed page 3–24 contains the following typos: – 1st line: "cvonents" should be "components" – 1st line of 1st paragraph: "of the le first" should be "of the list are first" – 5th line of 2nd paragraph: "ycorresponding" should be "corresponding" – 6th line of 2nd paragraph: "arrayype" should be "array type".
04.01.01	The header of page 4–2 should not be "Names and Expressions" but "Ada Reference Manual" justified at the right edge of the page.
10.04	The printed page 10–10 contains the following typos in the 3rd paragraph. – "prngram" should be "programm" – After "other program" in the 2nd line, the following words should be found: "libraries. Finally, there should be commands for interrogating the status of the units of a program library. The form of the commands" – suppress "nds" at the beginning of 3rd line.
14.01.02	In the first line of the 2nd paragraph after TRUNCATE, "phys" should be "physical".
C	In lower case letters, change 'A' into 'a' and 'Z' into 'z'.

Foreword

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 DoD requirements were formalized in a series of documents which were extensively reviewed by the Services, industrial organizations, universities, and foreign military departments. The culmination of that process was the Steelman Report to which the Ada language has been designed.

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-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, M.W. Davis, G. Ferran, 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 is the development of a formal definition using denotational semantics, with the participation of V. Donzeau-Gouge, G. Kahn and B. Lang. The other is 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 and W.L. Heimerdinger. H.G. Schmitz served as program manager.

Over the three years spent on this project, five intense one-week design reviews were conducted with the participation of H. Harte, A.L. Hisgen, P. Knueven, M. Kronental, G. Seegmueller, V. Stenning, and also F. Belz, P. Cohen, R. Converse, K. Correll, R. Dewar, A. Evans, 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 P. Belmont, E. Boebert, P. Bonnard, R. Brender, B. Brosgol, H. Clausen, M. Cox, T. Froggatt, H. Ganzinger, C. Hewitt, S. Kamin, J.L. Mansion, F. Minel, T. Phinney, J. Roehrich, V. Schneider, A. Singer, D. Slosberg, I.C. Wand, the reviewers of the group Ada-Europe, and the reviewers of the Tokyo study group assembled by N. Yoneda, E. Wada, and K. Kakehi.

These reviews and comments, the numerous evaluation reports received at the end of the first and second phase, the more than nine hundred language issue reports, comments, and test and evaluation reports received from fifteen different countries during the third phase of the project, and the on-going work of the IFIP Working Group 2.4 on system implementation languages and that of LTPL-E of Purdue Europe, 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 iteration of language requirements documents, leading to the Steelman specification.

This language definition was developed by Cii Honeywell Bull and Honeywell Systems and Research Center under contract to the United States Department of Defense. William E. Carlson served as the technical representative of the 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

Appendices

1. Introduction

This report describes the programming language Ada, designed in accordance with the Steelman requirements of the United States Department of Defense. Overall, the Steelman 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 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 program applications. This requires access to system dependent parameters and precise control over the representation of data. Finally, both application level and machine level input-output are defined.

1.1 Design Goals

Ada was designed with three overriding concerns: a recognition of the importance of program reliability and maintenance, a concern for programming as a human activity, and efficiency.

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

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 with an intuitive mapping on what the user will normally expect.

Like many other human activities, the development of programs is becoming more and 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 program units are directly related to this idea, which has ramifications in many other aspects of the language.

No language can avoid the problem of efficiency. Languages that require overly 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 required excessive machine resources was rejected.

Perhaps most importantly, none of the above goals was considered something that could be achieved after the fact. The design goals drove the entire design process from the beginning.

1.2 Language Summary

An Ada program is composed of one or more program units, which can be compiled separately. Program units may be subprograms (which define executable algorithms), packages (which define collections of entities), or tasks (which define concurrent computations). 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.

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

An Ada program will normally make use of a library of program units of general utility. The language provides means whereby individual organizations can construct their own libraries. To allow accurate control of program maintenance, the text of a separately compiled program unit must name the library units it requires.

Program units.

A subprogram is the basic unit for expressing an algorithm. There are two kinds of subprograms: procedures and functions. A procedure is the logical counterpart to a series of actions. For example, it may read in 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 logical counterpart to 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 is the basic unit for defining a sequence of actions that may be executed in parallel with other similar units. Parallel 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 object or a task type defining similar task objects.

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, and tasks 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 arguments provided at the call with the corresponding formal parameters of the subprogram.

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 basic iterative mechanism in the language is the loop statement. A loop statement specifies that a sequence of statements is to be executed repeatedly until an iteration clause is completed or an exit statement is encountered.

A block 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 is written as a procedure call; 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. A select statement allows a selective wait for one of several alternative rendezvous. Other forms of the select statement allow conditional or timed entry calls.

Execution of a program unit may lead to exceptional 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 these situations, the statements of a program unit can be textually followed by exception handlers describing the actions to be taken when the exceptional situation arises. Exceptions can be raised explicitly by a raise statement.

Data Types

Every object in the language has a type which characterizes a set of values and a set of applicable operations. There are four classes of types: scalar types (comprising enumeration and numeric types), composite types, access types, and private types.

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.

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

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.

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

Access types allow the construction of linked data structures created by the execution 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.

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

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

Other Facilities

Representation specifications can be used to specify the mapping between data types and features of an underlying machine. For example, the user can specify that objects of a given type must be represented with a specified number of bits, or that the components of a record are to be represented in a specified storage layout. Other features allow the controlled use of low level, non portable, or implementation dependent aspects, including the direct insertion of machine code.

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.

1.3 Sources

A continual 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 much lesser degree than usual because of the Steelman requirements. These requirements often simplified the design process by permitting us to concentrate on the design of a given system satisfying a well defined set of capabilities, rather than on the definition of the capabilities themselves.

Another significant simplification of our 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 a counterpart in Ada. We may say that whereas these previous designs could be considered as genuine research efforts, the language Ada is the result of a project in language design engineering, in an attempt to develop a product that represents the current state of the art.

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 the Pascal family.

Finally, the evaluation reports received on the initial formulation of the Green language, the Red, Blue and Yellow language proposals, the language reviews that took place at different stages of this project, and the more than nine hundred reports received from fifteen different countries on the preliminary definition of Ada, all had a significant impact on the final definition of the language.

1.4 Syntax Notation

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

(a) Lower case words, some containing embedded underscores, denote syntactic categories, for example

    adding_operator

(b) Boldface words denote reserved words, for example

    array

(c) Square brackets enclose optional items, for example

    end [identifier];

(d) Braces enclose a repeated item. The item may appear zero or more times. Thus an identifier list is defined by

    identifier_list ::= identifier {, identifier}

(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

(f) Any syntactic category prefixed by an italicized word and an underscore is equivalent to the unprefixed corresponding category name. The prefix is intended to convey some semantic information. For example *type*_name and *task*_name are both equivalent to the category name.