

ANSI/IEEE770X3.97-1983

American National Standard Pascal Computer Programming Language

An American National Standard

**IEEE Standard Pascal
Computer Programming Language**



Published by
The Institute of Electrical and Electronics Engineers, Inc



Distributed in cooperation with
Wiley-Interscience, a division of John Wiley & Sons, Inc

ANSI/IEEE
770 X3.97-1983

An American National Standard

**IEEE Standard Pascal
Computer Programming Language**

Joint Sponsors
**IEEE Pascal Standards Committee
of the
IEEE Computer Society
and
ANSI/X3J9 of
American National Standards Committee X3**

Approved September 17, 1981
IEEE Standards Board

Approved December 16, 1982
American National Standards Institute

ISBN 0-471-88944-X

Library of Congress Catalog Number 82-84259

© Copyright 1979

Institute of Electrical and Electronics Engineers, Inc

© Copyright 1983

American National Standards Institute, Inc
and
Institute of Electrical and Electronics Engineers, Inc

*No part of this publication may be reproduced in any form,
in an electronic retrieval system or otherwise,
without the prior written permission of the publisher.*

Published by

The Institute of Electrical and Electronics Engineers, Inc
345 East 47th Street, New York, NY 10017

January 7, 1983

SH08912

Foreword

(This Foreword is not a part of ANSI/IEEE770X3.97-1983, IEEE Standard Pascal Computer Programming Language.)

This standard provides an unambiguous and machine independent definition of the language Pascal. Its purpose is to facilitate portability of Pascal programs for use on a wide variety of data processing systems.

Language History.

The computer programming language Pascal was designed by Professor Niklaus Wirth to satisfy two principal aims:

(1) To make available a language suitable for teaching programming as a systematic discipline based on certain fundamental concepts clearly and naturally reflected by the language.

(2) To define a language whose implementations could be reliable and efficient on then available computers.

However, it has become apparent that Pascal has attributes that go far beyond these original goals. It is now being increasingly used commercially in the writing of system and application software. This standard is primarily a consequence of the growing commercial interest in Pascal and the need to promote the portability of Pascal programs between data processing systems.

In drafting this standard the continued stability of Pascal has been a prime objective. However, apart from changes to clarify the specification, one major change has been introduced. The syntax used to specify procedural and functional parameters has been changed to require the use of a procedure or function heading, as appropriate (see 6.6.3.1); this change was introduced to overcome a language insecurity.

Project History.

(Throughout, the term JPC refers to the Joint ANSI/X3J9-IEEE Pascal Standards Committee.) In the fall of 1978 application was made to the IEEE Standards Board by the IEEE Computer Society to authorize project P770. After approval, the first meeting was held in January 1979.

In December of 1978, X3J9 convened as a result of a SPARC resolution to form a US TAG (Technical Advisory Group) for the ISO Pascal standardization effort initiated by the UK.

In agreement with IEEE representatives, in February of 1979, an X3 resolution combined the X3J9 and P770 committees into a single committee called the Joint ANSI/X3J9-IEEE Pascal Standards Committee. The first meeting as JPC was held in April 1979.

At its 7th meeting in April 1980, JPC reviewed the first draft of ISO dp7185 and submitted its technical comments as JPC/80-111 together with their recommendation to X3 that the US should vote "no, because" in light of these comments.

For a summary of other important events in the project's history, see the annual reports from 1979 through 1981. Two other significant events occurred in February 1981:

(1) JPC reviewed the second draft of ISO dp7185 and submitted its technical comments as JPC/81-42 together with their recommendation to X3 that the US should vote "yes, but" in light of those comments.

(2) JPC passed a motion to submit to TC letter ballot the ISO dp7185 second draft together with JPC/81-42 and JPC/81-41 which describes how the requirements for compliance with the dpANS and proposed IEEE Pascal standard differ from compliance with the ISO draft proposed standard.

Project Charter.

As X3J9, it approved SD-3 described its program of work to be:

(1) Maintain a liaison with the ISO, BSI, and IEEE committees to work toward a common working draft standard. This work should include review of those bodies' documents and forwarding of comments based on that review. The eventual draft proposed American National Standard Pascal shall be compatible with any ISO Pascal standard and identical in content with the jointly developed proposed IEEE Pascal standard.

(2) Provide a means for review of all Pascal standardization activities.

(3) Carry out the development of a Pascal standard.

(4) Identify and evaluate common existing practices in the area of Pascal extensions.

(5) Act as a liaison group with organizations interested in interpretation of American National Standard Pascal.

The resolution to form JPC clarified the dual function of the single joint committee to produce a dpANS and a proposed IEEE Pascal standard, identical in content.

Technical Development.

(1) *Technical Constraints by X3 or IEEE.* None.

(2) *Technical Alternatives Considered.* In view of the charter to maintain compatibility with "any ISO Pascal standard," technical alternatives, suggestions, and recommendations were submitted with US letter ballots on ISO draft proposals. These items were then resolved by Working Group 4 as part of the international standardization process.

(3) *Significant Technical Issues on which JPC was Initially Divided.* Divergent opinions on the technical issues concerning conformant arrays were forwarded as part of the US letter ballot comments (JPC/80-111, JPC/81-42). To resolve the issue within the JPC, the conformant arrays definition was removed from the proposed domestic standard. Therefore, extensions to the language that solve the conformant array issue are permitted in the same manner as any other extension.

(4) Information Collection Techniques Employed to Obtain Broader Input Data.

Working Draft 3 published April 1979 in IEEE Computer magazine

Working Draft 3 published January 1979 in Pascal News No 14

dp7185 First Draft published April 1980 in SIGPLAN Notices

dp7185 First Draft published May 1980 in Pascal News No 18

dp7185 Second Draft published December 1980 in Pascal News No 20

Comments received from public review were distributed to all JPC members and discussed as part of the technical review. These were also forwarded directly to the appropriate international body responsible for an ISO dp.

(5) Related Domestic Effort. X3J9 (an X3 Technical Committee) and IEEE P770 combined to form the Joint ANSI/X3J9-IEEE Pascal Standards Committee in early 1979. The joint collaboration contributed in the international development of ISO/7185, from which this standard was developed.

International.

The British Standards Institution (BSI) Committee OIS/5 has been our counterpart in the UK. They are the sponsoring body for the ISO dp for Pascal. There are similar active groups in Australia, Canada, France, Germany, Netherlands and Japan. The international liaisons on these committees have been kept informed of the US activity by being on the JPC mailing list. Also, these committees have assigned members to WG4, the international working group for Pascal.

Other Standards and their Relationship to an American National Standard and IEEE Pascal Standard.

ISO Pascal Standard. See the preceding section on **Project Charter** and the following section on **Differences**.

Differences of Technical Substance Between this Standard and the International Standard, as Represented by ISO/DIS 7185.

The differences of technical substance are:

(1) The domestic standard does not include the conformant array feature.

(2) The domestic standard specifies that extensions may alter the status of implementation-dependent features or errors. The ISO document prohibits extensions from altering the status of these items.

(3) The domestic standard specifies that the relationship, if any, between end-of-line and values of the char-type shall be implementation-dependent. The ISO document requires that end-of-line not be a value of the char-type.

(4) This standard specifies the ordering of evaluation, accessing, and binding of the parameters of **read**, **readln**, **write**, and **writeln**. The international standard does not address this areas, leaving it neither specified, implementation-defined, nor implementation-dependent.

At the time of adoption of this standard, the text of the international standard was observed to contain several errors of definition and several

points of apparent technical ambiguity or lack of clarity. This standard embodies corrections to those errors, and embodies certain wording which clarifies the apparent ambiguities. These differences in manner of definition are not differences of technical substance.

The errors that are corrected in this standard are in the definition of:

- (1) String-element alternatives
- (2) Data-transfer procedure parameters types
- (3) Lexicographic ordering
- (4) Control-variable usage restrictions

The international standard appears to permit or require the repeated evaluation of the file that is referenced by I/O procedures and of the arrays that are referenced by the data-transfer procedures. A detailed analysis is required to determine that the international standard does not in fact permit such repeated evaluation. This standard states that requirement more explicitly.

Except as noted in the list above of *Differences of Technical Substance*, compliance with this standard is equivalent to compliance at level 0 with the international standard in the following sense:

(1) Any program complying with this standard complies at level 0 with the international standard

(2) Any program or processor complying at level 0 with the international standard complies with this standard

(3) Any processor complying at level 1 with the international standard complies with this standard if it is “able to process in a manner similar to that specified for errors any use of [the conformant-array feature]”

(4) Any processor complying with this standard and not providing an extension which covers part or all of the intent of the conformant-array feature complies at level 0 with the international standard

(5) Any processor complying with this standard and providing an extension covering part or all of the intent of the conformant-array feature complies at level 1 with the international standard if it also includes the conformant-array feature.

Additionally, a program which uses any extension does not comply with either standard.

Future Work.

An SD3 for extended Pascal has been approved by X3 to authorize future work.

Suggestions for the improvement of this standard are welcomed. These suggestions should be sent to

Secretary
IEEE Standards Board
345 East 47th Street
New York, NY 10017

When the IEEE Standards Board approved this standard on September 17, 1981, it had the following membership:

I. N. Howell, Jr., *Chairman*

Irving Kolodny, *Vice Chairman*

Sava I. Sherr, *Secretary*

G. Y. R. Allen
J. J. Archambault
H. H. Beall
J. T. Boettger
Edward Chelotti
Edward J. Cohen
Len S. Corey

Jay Forster
Kurt Greene
Loering M. Johnson
Joseph L. Koepfinger
J. E. May
Donald T. Michael
J. P. Riganati

F. Rosa
R. W. Seelbach
J. S. Stewart
W. E. Vannah
Virginus N. Vaughan, Jr
Art Wall
Robert E. Weiler

*Member emeritus

This standard was processed and approved for submittal to ANSI by the American National Standards Committee on Information Systems, X3. Committee approval of this standard does not necessarily imply that all committee members voted for its approval.

At the time it approved this standard, the X3 committee had the following members:

John F. Auwaerter, *Chairman*

J.A.N. Lee, *Vice Chairman*

Catherine A. Kachurik, *Secretary*

<i>Organization</i>	<i>Name of Representative</i>
AMP Incorporated.	Patrick E. Lannan C. Brill (<i>Alt</i>)
American Bankers Association.	Andrew Ernst Chris Crawford (<i>Alt</i>)
American Express Co.	R. S. Newman R. G. Wilson (<i>Alt</i>)
American Library Association	Paul Peters
American Nuclear Society.	Geraldine C. Main D. R. Vondy (<i>Alt</i>)
Association for Computer Machinery	J. A. N. Lee Pat Skelly (<i>Alt</i>)
Association of American Railroads	R. A. Petrash
Association of Computer Users	Hillel Segal Thomas Kurihara (<i>Alt</i>)
Burroughs Corporation.	Ira R. Purchis Jerrold S. Foley (<i>Alt</i>)
Control Data Corporation.	Charles E. Cooper Keith Lucke (<i>Alt</i>)
Data General Corporation.	Steven W. Weingart Anthony M. Goschalk (<i>Alt</i>)
Data Processing Management Association.	Ardyn E. Dubnow Joseph A. Federici (<i>Alt</i>)
Digital Equipment Computers Users Society.	James Hodges John R. Barr (<i>Alt</i>)

<i>Organization</i>	<i>Name of Representative</i>
Digital Equipment Corporation	Lois C. Frampton Gary S. Robinson (<i>Alt</i>)
GUIDE International	Frank Kirshenbaum Leland Milligan (<i>Alt</i>)
General Services Administration	William C. Rinehuls Donald J. Page (<i>Alt</i>)
Harris Corporation	Sam Mathan David Abmayr (<i>Alt</i>)
Hewlett-Packard	Donald C. Loughry
Honeywell Information Systems	Thomas J. McNamara Alan Teubner (<i>Alt</i>)
IBM Corporation	Mary Anne Gray J. S. Wilson (<i>Alt</i>)
IEEE Communications Society	Thomas A. Varetoni
IEEE Computer Society	Robert Poston Robert S. Stewart (<i>Alt</i>)
Lawrence Berkeley Laboratory	James A. Baker Robert J. Harvey (<i>Alt</i>)
Life Office Management Association	John I. Burke James F. Foley, Jr (<i>Alt</i>)
3M Company	R. C. Smith
Moore Business Forms	D. H. Oddy
NCR Corporation	Thomas W. Kern William E. Synder (<i>Alt</i>)
National Bureau of Standards	Robert E. Rountree James H. Burrows (<i>Alt</i>)
National Communications System	Marshall L. Cain George W. White (<i>Alt</i>)
Perkin-Elmer Corporation	David Ellis David Saunders (<i>Alt</i>)
Prime Computer	Jeffrey C. Flowers Winfried A. Burke (<i>Alt</i>)
Professional Secretaries International	Jerome Heitman P. E. Pesce (<i>Alt</i>)
Recognition Technology Users Association	Herbert F. Schantz G. W. Wetzel (<i>Alt</i>)
SHARE, Inc.	Thomas B. Steel Daniel Schuster (<i>Alt</i>)
Society of Certified Data Processors	Thomas M. Kurihara Ardyn E. Dubnow (<i>Alt</i>)
Sperry Univac	Marvin W. Bass Charles D. Card (<i>Alt</i>)
Telephone Group	Henry L. Marchese J. A. Owen (<i>Alt</i>) Stewart M. Garland (<i>Alt</i>)
Texas Instruments, Inc.	Presley Smith Don Caraway (<i>Alt</i>)
Travelers Insurance Companies, Inc.	Joseph T. Brophy
US Department of Defense	William LaPlant Harry Pontius (<i>Alt</i>)
Wang Laboratories, Inc.	Carl W. Schwarcz Marsha Hayek (<i>Alt</i>)
Xerox Corporation	John L. Wheeler Arthur R. Machell (<i>Alt</i>)

The Joint ANSI/X3J9-IEEE Pascal Standards Committee, which developed this standard had the following members:

Carol Sledge, *Chairman*

Michael P. Hagerty, *Vice Chairman*

David L. Reese, *Secretary*

Joe Cointment, *International Representative*

Michael Alexander
Jeffrey Allen
Ed Barkmeyer
W. Ashby Boaz
A. Windsor Brown
Jerry R. Brookshire
Tomas M. Burger
David S. Cargo
Richard J. Cichelli
Roger Cox
Jean Danver
Debra Deutsch
Bob Dietrich
Victor A. Falwarczny
G. G. Gustafson
Thomas Giventer
Hellmut Golde
David N. Gray
Paul Gregory
Charles E. Haynes
Christopher Henrich
Steven Hiebert
Ruth Higgins
Charles Hill

Steven Hobbs
Albert A. Hoffman
Robert Hutchins
Rosa C. Hwang
Scott Jameson
David Jones
Steen Jurs
Mel Kanner
John Kaufmann
Leslie Klein
Bruce Knobe
Dennis Kodimer
Ronald E. Kole
Alan A. Kortesoja
Edward Krall
Robert Lange
Rainer McCown
Jim Miner
Eugene N. Miya
Mark Molloy
Dennis Nicholson
Mark Overgaard
Ted C. Park
Donald D. Peckham
David Peercy

Robert Poon
David L. Presberg
William C. Price
Bruce Ravenel*
David C. Robbins
Lynne Rosenthal
Tom Rudkin
Stephen C. Schwarm
Rick Shaw
Barry Smith
Rudeen S. Smith
Bill Stackhouse
Marius Troost**
Thomas N. Turba
Prescott K. Turner
Howard Turtle
Robert Tuttle
Richard C. Vile, Jr
Larry B. Weber
David Weil
Thomas R. Wilcox
Thomas Wolfe
Harvy Wohlwend
Kenneth M. Zemrowski

*Past Chairman IEEE Pascal Committee

**Past Chairman X3J9 Committee

Others who contributed to the development of this standard are:

A. M. Addyman: Chairman BSI OIS/5 and Convener of ISO/TC 97/SC 5/
Working Group 4 Pascal

Members of ISO/TC 97/SC 5/Working Group 4 Pascal

Thomas N. Turba and Sperry Univac made major contributions to the publication of this edition by editing the approved draft and supplying type-set copy. Their assistance is acknowledged with gratitude.

IEEE Standards documents are developed within the Technical Committees of the IEEE Societies and the Standards Coordinating Committees of the IEEE Standards Board. Members of the committees serve voluntarily and without compensation. They are not necessarily members of the Institute. The standards developed within IEEE represent a consensus of the broad expertise on the subject within the Institute as well as those activities outside of IEEE which have expressed an interest in participating in the development of the standard.

Use of an IEEE Standard is wholly voluntary. The existence of an IEEE Standard does not imply that there are no other ways to produce, test, measure, purchase, market, or provide other goods and services related to the scope of the IEEE Standard. Furthermore, the viewpoint expressed at the time a standard is approved and issued is subject to change brought about through developments in the state of the art and comments received from users of the standard. Every IEEE Standard is subjected to review at least once every five years for revision or reaffirmation. When a document is more than five years old, and has not been reaffirmed, it is reasonable to conclude that its contents, although still of some value, do not wholly reflect the present state of the art. Users are cautioned to check to determine that they have the latest edition of any IEEE Standard.

Comments for revision of IEEE Standards are welcome from any interested party, regardless of membership affiliation with IEEE. Suggestions for changes in documents should be in the form of a proposed change of text, together with appropriate supporting comments.

Interpretations: Occasionally questions may arise regarding the meaning of portions of standards as they relate to specific applications. When the need for interpretations is brought to the attention of IEEE, the Institute will initiate action to prepare appropriate responses. Since IEEE Standards represent a consensus of all concerned interests, it is important to ensure that any interpretation has also received the concurrence of a balance of interests. For this reason IEEE and the members of its technical committees are not able to provide an instant response to interpretation requests except in those cases where the matter has previously received formal consideration.

Comments on standards and requests for interpretations should be addressed to:

Secretary, IEEE Standards Board
345 East 47th Street
New York, NY 10017
USA

This standard was derived from the second revision of ISO dp7185 and has been updated to include changes to the third revision of ISO dp7185 and responses to public comments on the first draft of ANSI/IEEE 770X3.97-1983.

EDITORIAL NOTE: It is the normal convention to use italic type for algebraic quantities. Since the status of such quantities contained in this standard may or may not directly represent true variable quantities, this convention has not been adopted in this standard.

Contents

SECTION	PAGE
1. Scope	19
2. References	19
3. Definitions	20
3.1 Error	20
3.2 Extension	20
3.3 Implementation-Defined	20
3.4 Implementation-Dependent	20
3.5 Processor	20
4. Definitional Conventions	21
5. Compliance	22
5.1 Processors	22
5.2 Programs	24
6. Requirements	25
6.1 Lexical Tokens	25
6.1.1 General	25
6.1.2 Special-Symbols	25
6.1.3 Identifiers	25
6.1.4 Directives	26
6.1.5 Numbers	26
6.1.6 Labels	27
6.1.7 Character-Strings	27
6.1.8 Token Separators	28
6.1.9 Lexical Alternatives	28
6.2 Blocks, Scope and Activations	29
6.2.1 Block	29
6.2.2 Scope	30
6.2.3 Activations	31
6.3 Constant-Definitions	33
6.4 Type-Definitions	33
6.4.1 General	33
6.4.2 Simple-Types	34
6.4.2.1 General	34
6.4.2.2 Required Simple-Types	34
6.4.2.3 Enumerated-Types	35
6.4.2.4 Subrange-Types	36
6.4.3 Structured-Types	36
6.4.3.1 General	36

SECTION	PAGE
6.4.3.2 Array-Types	37
6.4.3.3 Record-Types	39
6.4.3.4 Set-Types	42
6.4.3.5 File-Types	43
6.4.4 Pointer-Types	46
6.4.5 Compatible Types	46
6.4.6 Assignment-Compatibility	46
6.4.7 Example of a Type Definition-Part	47
6.5 Declarations and Denotations of Variables	48
6.5.1 Variable-Declarations	48
6.5.2 Entire-Variables	49
6.5.3 Component-Variables	49
6.5.3.1 General	49
6.5.3.2 Indexed-Variables	50
6.5.3.3 Field-Designators	51
6.5.4 Identified-Variables	52
6.5.5 Buffer-Variables	52
6.6 Procedure and Function Declarations	53
6.6.1 Procedure-Declarations	53
6.6.2 Function-Declarations	56
6.6.3 Parameters	59
6.6.3.1 General	59
6.6.3.2 Value Parameters	60
6.6.3.3 Variable Parameters	60
6.6.3.4 Procedural Parameters	61
6.6.3.5 Functional Parameters	61
6.6.3.6 Parameter List Congruity	61
6.6.4 Required Procedures and Functions	62
6.6.4.1 General	62
6.6.5 Required Procedures	62
6.6.5.1 General	62
6.6.5.2 File Handling Procedures	62
6.6.5.3 Dynamic Allocation Procedures	65
6.6.5.4 Transfer Procedures	66
6.6.6 Required Functions	67
6.6.6.1 General	67
6.6.6.2 Arithmetic Functions	67
6.6.6.3 Transfer Functions	68
6.6.6.4 Ordinal Functions	68
6.6.6.5 Boolean Functions	69
6.7 Expressions	69
6.7.1 General	69

SECTION	PAGE
6.7.2 Operators	71
6.7.2.1 General	71
6.7.2.2 Arithmetic Operators	72
6.7.2.3 Boolean Operators	74
6.7.2.4 Set Operators	74
6.7.2.5 Relational Operators	74
6.7.3 Function Designators	75
6.8 Statements	76
6.8.1 General	76
6.8.2 Simple-Statements	77
6.8.2.1 General	77
6.8.2.2 Assignment-Statements	77
6.8.2.3 Procedure-Statements	78
6.8.2.4 Goto-Statements	79
6.8.3 Structured-Statements	79
6.8.3.1 General	79
6.8.3.2 Compound-Statements	79
6.8.3.3 Conditional-Statements	80
6.8.3.4 If-Statements	80
6.8.3.5 Case-Statements	80
6.8.3.6 Repetitive-Statements	81
6.8.3.7 Repeat-Statements	81
6.8.3.8 While-Statements	82
6.8.3.9 For-Statements	82
6.8.3.10 With-Statements	85
6.9 Input and Output	86
6.9.1 The Procedure Read	86
6.9.2 The Procedure Readln	88
6.9.3 The Procedure Write	88
6.9.3.1 Write-Parameters	89
6.9.3.2 Char-Type	89
6.9.3.3 Integer-Type	90
6.9.3.4 Real-Type	90
6.9.3.4.1 Floating-Point Representation	90
6.9.3.4.2 Fixed-Point Representation	92
6.9.3.5 Boolean-Type	93
6.9.3.6 String-Types	93
6.9.4 The Procedure Writeln	93
6.9.5 The Procedure Page	94
6.10 Programs	94

TABLES	PAGE
Table 1 Metalanguage Symbols	21
Table 2 Dyadic Arithmetic Operations	72
Table 3 Monadic Arithmetic Operations	72
Table 4 Set Operations	74
Table 5 Relational Operations	75
APPENDIXES	
Appendix A Collected Syntax	99
Appendix B Index	109
Appendix C Required Identifiers	123
Appendix D Errors	124

An American National Standard

IEEE Standard Pascal

Computer Programming Language

1. Scope

1.1. This standard specifies the semantics and syntax of the computer programming language Pascal by specifying requirements for a processor and for a conforming program.

1.2. This standard does not specify:

(a) the size or complexity of a program and its data that will exceed the capacity of any specific data processing system or the capacity of a particular processor, nor the actions to be taken when the corresponding limits are exceeded;

(b) the minimal requirements of a data processing system that is capable of supporting an implementation of a processor for Pascal;

(c) the method of activating the program-block or the set of commands used to control the environment in which a Pascal program is transformed and executed;

(d) the mechanism by which programs written in Pascal are transformed for use by a data processing system;

(e) the method for reporting errors or warnings;

(f) the typographical representation of a program published for human reading.

2. Reference

ISO 646 : The 7-bit coded character set for information processing interchange.

3. Definitions

For the purposes of this standard, the following definitions apply.

NOTE: To draw attention to language concepts, some terms are printed in *italics* on their first mention in this standard.

3.1. Error. A violation by a program of the requirements of this standard that a processor is permitted to leave undetected.

NOTES:

(1) If it is possible to construct a program in which the violation or non-violation of this standard requires knowledge of the data read by the program or the implementation definition of implementation-defined features, then violation of that requirement is classified as an *error*. Processors may report on such violations of the requirement without such knowledge, but there always remain some cases that require execution or simulated execution, or proof procedures with the required knowledge. Requirements that can be verified without such knowledge are not classified as errors.

(2) Processors should attempt the detection of as many errors as possible, and to as complete a degree as possible. Permission to omit detection is provided for implementations in which the detection would be an excessive burden.

3.2. Extension. A modification to Section 6 of the requirements of this standard that does not invalidate any program complying with this standard, as defined by Section 5.2, except by prohibiting the use of one or more particular spellings of identifiers.

3.3. Implementation-Defined. Possibly differing between processors, but defined for any particular processor.

3.4. Implementation-Dependent. Possibly differing between processors and not necessarily defined for any particular processor.

3.5. Processor. A system or mechanism that accepts a program as input, prepares it for execution, and executes the process so defined with data to produce results.

NOTE: A processor may consist of an interpreter, a compiler and run-time system, or other mechanism, together with an associated host computing machine and operating system, or other mechanism for achieving the same effect. A compiler in itself, for example, does not constitute a processor.

4. Definitional Conventions

The metalanguage used in this standard to specify the syntax of the constructs is based on Backus-Naur Form. The notation has been modified from the original to permit greater convenience of description and to allow for iterative productions to replace recursive ones. Table 1 lists the meanings of the various metasymbols. Further specification of the constructs is given by prose and, in some cases, by equivalent program fragments. Any identifier that is defined in Section 6 as a required identifier shall denote the corresponding required entity by its occurrence in such a program fragment. In all other respects, any such program fragment is bound by any pertinent requirement of this standard.

Table 1.
Metalanguage Symbols

Metasymbol	Meaning
=	shall be defined to be
	alternatively
.	end of definition
[x]	0 or 1 instance of x
{ x }	0 or more instances of x
(x y)	grouping: either of x or y
"xyz"	the terminal symbol xyz
meta-identifier	a non-terminal symbol

A meta-identifier shall be a sequence of letters and hyphens beginning with a letter.

A sequence of terminal and non-terminal symbols in a production implies the concatenation of the text that they ultimately represent. Within 6.1 this concatenation is direct; no characters shall intervene. In all other parts of this standard the concatenation is in accordance with the rules set out in 6.1.

The characters required to form Pascal programs shall be those implicitly required to form the tokens and separators defined in 6.1.

Use of the words *of*, *in*, *containing* and *closest-containing* when expressing a relationship between terminal or non-terminal symbols shall have the following meanings.

the x *of* a y: refers to the x occurring directly in a production defining y.

the x *in* a y: is synonymous with 'the x *of* a y'.

a y *containing* an x: refers to any y from which an x is directly or indirectly derived.

the y *closest-containing* an x: that y which contains an x but does not contain another y *containing* that x.

These syntactic conventions are used in Section 6 to specify certain syntactic requirements and also the contexts within which certain semantic specifications apply.

5. Compliance

5.1. Processors. A processor complying with the requirements of this standard shall:

- (a) accept all the features of the language specified in Section 6 with the meanings defined in Section 6;
- (b) *(This section intentionally left blank to preserve numbering with ISO dp7185.)*
- (c) not require the inclusion of substitute or additional language elements in a program in order to accomplish a feature of the language that is specified in Section 6;
- (d) be accompanied by a document that provides a definition of all implementation-defined features;
- (e) be able to determine whether or not a program violates any requirement of this standard, where such a violation is not designated

an error, and report the result of this determination to the user of the processor; in the case where the processor does not examine the whole program, the user shall be notified that the determination is incomplete whenever no violations have been detected in the program text examined;

(f) treat each violation that is designated an error in at least one of the following ways:

- (1) there shall be a statement in an accompanying document that the error is not reported;
- (2) the processor shall report during preparation of the program for execution that an occurrence of that error was possible;
- (3) the processor shall report the error during preparation of the program for execution;
- (4) the processor shall report the error during execution of the program, and terminate execution of the program;

and if any violations that are designated as errors are treated in the manner described in 5.1(f)(1), then a note referencing each such treatment shall appear in a separate section of the accompanying document;

(g) be accompanied by a document that separately describes any features accepted by the processor that are prohibited or not specified in Section 6: such extensions shall be described as being 'extensions to Pascal as specified by ANSI/IEEE770X3.97-1983';

(h) be able to process in a manner similar to that specified for errors any use of any such extension;

(i) be able to process in a manner similar to that specified for errors any use of an implementation-dependent feature.

NOTES:

(1) The phrase 'be able to' is used in 5.1 to permit the implementation of a switch with which the user may control the reporting.

(2) In cases where the compilation is aborted due to some limitation of tables, etc., an incomplete determination of the kind 'No violations were detected, but the examination is incomplete.' will satisfy the requirements of Section 5.1(e). In a similar manner, an interpretive or direct execution processor may report an incomplete determination for a program of which all aspects have not been examined.

A processor that purports to comply, wholly or partially, with the requirements of this standard shall do so only in the following terms. A *compliance statement* shall be produced by the processor as a consequence of using the processor, or shall be included in accompanying documentation. If the processor complies in all respects with the requirements of this standard the compliance statement shall be:

<This processor> complies with the requirements of ANSI/IEEE770X3.97-1983.

If the processor complies with some but not all of the requirements of this standard then it shall not use the above statement, but shall instead use the following compliance statement:

<This processor> complies with the requirements of ANSI/IEEE770X3.97-1983 with the following exceptions:

<followed by a reference to, or a complete list of, the requirements of the standard with which the processor does not comply> .

In both cases the text *<This processor>* shall be replaced by an unambiguous name identifying the processor.

NOTE: Processors that do not comply fully with the requirements of the standard are not required to give full details of their failures to comply in the compliance statement; a brief reference to accompanying documentation that contains a complete list in sufficient detail to identify the defects is sufficient.

5.2. Programs. A program complying with the requirements of this standard shall:

- (a) use only those features of the language specified in Section 6,
- (b) not rely on any particular interpretation of implementation-dependent features.

NOTES:

(1) A program that complies with the requirements of this standard may rely on particular implementation-defined values or features.

(2) The requirements for compliant programs and compliant processors do not require that the results produced by a compliant program are always the same when processed by a compliant processor. They may be, or they may differ, depending on the program. A simple program to illustrate this is:

```
program x(output); begin writeln(maxint) end.
```

6. Requirements

6.1. Lexical Tokens.

NOTE: The syntax given in this subsection (6.1) describes the formation of lexical tokens from characters and the separation of these tokens, and therefore does not adhere to the same rules as the syntax in the rest of this standard.

6.1.1. General. The lexical tokens used to construct Pascal programs shall be classified into special-symbols, identifiers, directives, unsigned-numbers, labels and character-strings. The representation of any letter (upper-case or lower-case, differences of font, etc.) occurring anywhere outside of a character-string (see 6.1.7) shall be insignificant in that occurrence to the meaning of the program.

letter = "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" .

digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" .

6.1.2. Special-Symbols. The special-symbols are tokens having special meanings and shall be used to delimit the syntactic units of the language.

special-symbol = "+" | "-" | "*" | "/" | "=" | "<" | ">" | "[" |
"]" | "." | "," | ":" | ";" | "!" | "(" | ")" | "<>" |
"<=" | ">=" | ":" | ".." | word-symbol .

word-symbol = "and" | "array" | "begin" | "case" | "const" | "div" |
"do" | "downto" | "else" | "end" | "file" | "for" |
"function" | "goto" | "if" | "in" | "label" | "mod" |
"nil" | "not" | "of" | "or" | "packed" | "procedure" |
"program" | "record" | "repeat" | "set" | "then" |
"to" | "type" | "until" | "var" | "while" | "with" .

6.1.3. Identifiers. Identifiers may be of any length. All characters of an identifier shall be significant in distinguishing between identifiers. No identifier shall have the same spelling as any word-symbol. Identifiers that are specified to be *required* shall have special significance (see 6.2.2.10 and 6.10).

identifier = letter { letter | digit } .

EXAMPLES:

```
X time readinteger WG4 AlterHeatSetting
  InquireWorkstationTransformation
  InquireWorkstationIdentification
```

6.1.4. Directives. A directive shall occur only *in* a procedure-declaration or function-declaration. The directive **forward** shall be the only required directive (see 6.6.1 and 6.6.2). No directive shall have the same spelling as any word-symbol.

directive = letter { letter | digit } .

NOTE: Many processors provide, as an extension, the directive **external**, which is used to specify that the procedure-block or function-block corresponding to that procedure-heading or function-heading is external to the program-block. Usually it is in a library in a form to be input to, or that has been produced by, the processor.

6.1.5. Numbers. An unsigned-integer shall denote in decimal notation a value of integer-type (see 6.4.2.2). An unsigned-real shall denote in decimal notation a value of real-type (see 6.4.2.2). The letter 'e' preceding a scale factor shall mean *times ten to the power of*. The value denoted by an unsigned-integer shall be in the closed interval 0 to **maxint** (see 6.4.2.2 and 6.7.2.2).

signed-number = signed-integer | signed-real .

signed-real = [sign] unsigned-real .

signed-integer = [sign] unsigned-integer .

unsigned-number = unsigned-integer | unsigned-real .

sign = "+" | "-" .

unsigned-real =
 unsigned-integer "." fractional-part ["e" scale-factor] |
 unsigned-integer "e" scale-factor .

unsigned-integer = digit-sequence .

fractional-part = digit-sequence .

scale-factor = signed-integer .

digit-sequence = digit { digit } .

EXAMPLES:

1e10 1 +100 -0.1 5e-3 87.35E+8

6.1.6. Labels. Labels shall be digit-sequences and shall be distinguished by their apparent integral values and shall be in the closed interval 0 to 9999.

label = digit-sequence .

6.1.7. Character-Strings. A character-string *containing* a single string-element shall denote a value of the required char-type (see 6.4.2.2). A character-string *containing* more than one string-element shall denote a value of a string-type (see 6.4.3.2) with the same number of components as the character-string *contains* string-elements. There shall be an implementation-defined one-to-one correspondence between the set of alternatives from which string-elements are drawn and a subset of the values of the required char-type. The occurrence of a string-element in a character-string shall denote the occurrence of the corresponding value of char-type.

character-string = "" string-element { string-element } "" .

string-element = apostrophe-image | string-character .

apostrophe-image = "'"

string-character = one-of-a-set-of-implementation-defined-characters .

NOTE: Conventionally, the apostrophe-image is regarded as a substitute for the apostrophe character, which cannot be a string-character.

EXAMPLES:

'A' ';' ''''
'Pascal' 'THIS IS A STRING'

6.1.8. Token Separators. The construct

"{" any-sequence-of-characters-and-separation-of-lines-not-containing-right-brace "}"

shall be a comment if the / does not occur within a character-string or within a comment. The substitution of a space for a comment shall not alter the meaning of a program.

Comments, spaces (except in character-strings), and the separation of consecutive lines shall be considered to be token separators. Zero or more token separators may occur between any two consecutive tokens, or before the first token of a program text. There shall be at least one separator between any pair of consecutive tokens made up of identifiers, word-symbols, labels or unsigned-numbers. No separators shall occur within tokens.

6.1.9. Lexical Alternatives. The representation for lexical tokens and separators given in 6.1.1 to 6.1.8 shall constitute a *reference representation* for these tokens and separators. The reference representation shall be used for program interchange.

To facilitate the use of Pascal on processors that do not support the reference representation, the following alternatives have been defined. All processors that have the required characters in their character set shall provide both the reference representations and the alternative representations, and the corresponding tokens or separators shall not be distinguished.

The alternative representations for the tokens shall be:

Reference token	Alternative token
↑	@
[	(.
]	.)

NOTE: The character ^ that appears in some national variants of ISO 646 is regarded as identical to the character ↑. In this standard, the character ↑ has been used because of its greater visibility.

The alternative forms of comment shall be all forms of comment where one or both of the following substitutions are made:

Delimiting character	Alternative delimiting pair of characters
{	(*
}	*)

NOTES:

(1) A comment may thus commence with / and end with */, or commence with /* and end with /.

(2) If the sequence /* occurs in a comment, it is equivalent to // and marks the end of the comment, because the substitution is only for a delimiting character.

(3) See also 1.2(f).

6.2. Blocks, Scope and Activations.

6.2.1. Block.

A block *closest-containing* a label-declaration-part *in* which a label occurs shall *closest-contain* exactly one statement *in* which that label occurs. The occurrence of a label *in* the label-declaration-part *of* a block shall be its defining-point as a label for the region that is the block.

```
block = label-declaration-part
      constant-definition-part
      type-definition-part
      variable-declaration-part
      procedure-and-function-declaration-part
      statement-part .
```

```
label-declaration-part = [ "label" label { "," label } ";" ] .
```

```
constant-definition-part =
  [ "const" constant-definition ";" { constant-definition ";" } ] .
```

```
type-definition-part =
  [ "type" type-definition ";" { type-definition ";" } ] .
```

```
variable-declaration-part =
  [ "var" variable-declaration ";" { variable-declaration ";" } ] .
```

```
procedure-and-function-declaration-part =
  { ( procedure-declaration | function-declaration ) ";" } .
```

The statement-part shall specify the algorithmic actions to be executed upon an activation of the block.

statement-part = compound-statement .

6.2.2. Scope.

6.2.2.1. Each identifier or label *contained* by the program-block shall have a defining-point.

6.2.2.2. Each defining-point shall have a region that is a part of the program text, and a scope that is a part or all of that region.

6.2.2.3. The region of each defining-point is defined elsewhere (see 6.2.1, 6.2.2.10, 6.3, 6.4.1, 6.4.2.3, 6.4.3.3, 6.5.1, 6.5.3.3, 6.6.1, 6.6.2, 6.6.3.1, 6.8.3.10, 6.10).

6.2.2.4. The scope of each defining-point shall be its region (including all regions enclosed by that region) subject to 6.2.2.5 and 6.2.2.6.

6.2.2.5. When an identifier or label has a defining-point for region A and another identifier or label having the same spelling has a defining-point for some region B enclosed by A, then region B and all regions enclosed by B shall be excluded from the scope of the defining-point for region A.

6.2.2.6. The region that is the field-specifier *of* a field-designator shall be excluded from the enclosing scopes.

6.2.2.7. When an identifier or label has a defining-point for a region, another identifier or label with the same spelling shall not have a defining-point for that region.

6.2.2.8. Within the scope of a defining-point of an identifier or label, each occurrence of an identifier or label having the same spelling as the identifier or label of the defining-point shall be designated an applied occurrence of the identifier or label of the defining-point, except for an occurrence that constituted the defining-point of that identifier or label; such an occurrence shall be designated a defining occurrence. No occurrence outside that scope shall be an applied occurrence.

NOTE: Within the scope of a defining-point of an identifier or label, there are no applied occurrences of an identifier or label that cannot be distinguished from it and have a defining-point for a region enclosing that scope.

6.2.2.9. The defining-point of an identifier or label shall precede all applied occurrences of that identifier or label *contained* by the program-block with one exception, namely that an identifier may have an applied occurrence in the type-identifier *of* the domain-type *of* any new-pointer-types *contained* by the type-definition-part that *contains* the defining-point of the type-identifier.

6.2.2.10. Identifiers that denote required constants, types, procedures and functions shall be used as if their defining-points have a region enclosing the program (see 6.1.3, 6.3, 6.4.1 and 6.6.4.1).

NOTE: The required identifiers **input** and **output** are not included, since these denote variables.

6.2.2.11. Whatever an identifier or label denotes at its defining-point shall be denoted at all applied occurrences of that identifier or label.

NOTE: Within syntax definitions, an applied occurrence of an identifier is qualified, e.g. type-identifier, whereas a use that constitutes a defining-point is not qualified.

6.2.3. Activations.

6.2.3.1. A procedure-identifier or function-identifier having a defining-point for a region that is a block within the procedure-and-function-declaration-part of that block shall be designated *local* to that block.

6.2.3.2. The activation of a block shall contain:

- (a) for the statement-part *of* the block, an algorithm, the completion of which shall terminate the activation (see also 6.8.2.4);
- (b) for each label *in* a statement having a defining-point in the label-declaration-part of the block, a program-point in the algorithm of the activation at that statement;
- (c) for each variable-identifier having a defining-point for the region that is the block, a variable possessing the type associated with the variable-identifier;
- (d) for each procedure-identifier local to the block, a procedure with the procedure-block corresponding to the procedure-identifier, and the formal parameters of that procedure-block;

(e) for each function-identifier local to the block, a function with the function-block corresponding to, and the result type associated with, the function-identifier, and the formal parameters of that function-block;

(f) if the block is a function-block, a result possessing the associated result type.

NOTE: Each activation contains its own algorithm, set of program-points, set of variables, set of procedures, and set of functions, distinct from every other activation.

6.2.3.3. The activation of a procedure or function shall be the activation of the block of its procedure-block or function-block, respectively, and shall be designated as *within* :

(a) the activation containing the procedure or function; and

(b) all activations that that containing activation is within.

NOTE: An activation of a block B can only be within activations of blocks containing B. Thus an activation is not within another activation of the same block.

Within an activation, an applied occurrence of a label or variable-identifier, or of a procedure-identifier or function-identifier local to the block of the activation, shall denote the corresponding program-point, variable, procedure, or function, respectively, of that activation; except that the function-identifier *of* an assignment-statement shall, within an activation of the function denoted by that function-identifier, denote the result of that activation.

6.2.3.4. A procedure-statement or function-designator contained in the algorithm of an activation and that specifies the activation of a block shall be designated the activation-point of that activation of the block.

6.2.3.5. All variables contained by an activation, except for those listed as program-parameters, and any result of an activation, shall be totally-undefined at the commencement of that activation. The algorithm, program-points, variables, procedures and functions, if any, shall exist until the termination of the activation.

6.3. Constant-Definitions. A constant-definition shall introduce an identifier to denote a value.

constant-definition = identifier "=" constant .

constant = [sign] (unsigned-number | constant-identifier) |
character-string .

constant-identifier = identifier .

The occurrence of an identifier *in* a constant-definition *of* a constant-definition-part *of* a block shall constitute its defining-point for the region that is the block. The constant *in* a constant-definition shall not contain an applied occurrence of the identifier *in* the constant-definition. Each applied occurrence of that identifier shall be a constant-identifier and shall denote the value denoted by the constant *of* the constant-definition. A constant-identifier *in* a constant containing an

occurrence of a sign shall have been defined to denote a value of real-type or of integer-type. The required constant-identifiers shall be as specified in 6.4.2.2 and 6.7.2.2.

6.4. Type-Definitions.

6.4.1. General. A type-definition shall introduce an identifier to denote a type. Type shall be an attribute that is possessed by every value and every variable. Each occurrence of a new-type shall denote a type that is distinct from any other new-type.

type-definition = identifier "=" type-denoter .

type-denoter = type-identifier | new-type .

new-type = new-ordinal-type |
new-structured-type |
new-pointer-type .

The occurrence of an identifier *in* a type-definition *of* a type-definition-part *of* a block shall constitute its defining-point for the region that is the block. Each applied occurrence of that identifier shall be a type-identifier and shall denote the same type as that which is denoted

by the type-denoter *of* the type-definition. Except for applied occurrences in the domain-type *of* a new-pointer-type, the type-denoter shall not contain an applied occurrence of the identifier *in* the type-definition.

Types shall be classified as simple, structured or pointer types. The required type-identifiers and corresponding required types shall be as specified in 6.4.2.2 and 6.4.3.5.

simple-type-identifier = type-identifier .

structured-type-identifier = type-identifier .

pointer-type-identifier = type-identifier .

type-identifier = identifier .

A type-identifier shall be considered as a simple-type-identifier, a structured-type-identifier, or a pointer-type-identifier, according to the type that it denotes.

6.4.2. Simple-Types.

6.4.2.1. General. A simple-type shall determine an ordered set of values. The values of each ordinal-type shall have integer ordinal numbers. An ordinal-type-identifier shall denote an ordinal-type.

simple-type = ordinal-type | real-type-identifier .

ordinal-type = new-ordinal-type | ordinal-type-identifier .

new-ordinal-type = enumerated-type | subrange-type .

ordinal-type-identifier = type-identifier .

real-type-identifier = type-identifier .

6.4.2.2. Required Simple-Types. The following types shall exist:

(a) *integer-type*. The required ordinal-type-identifier **integer** shall denote the integer-type. The values shall be a subset of the whole numbers, denoted as specified in 6.1.5 by signed-integer (see also 6.7.2.2). The ordinal number of a value of integer-type shall be the value itself.

(b) *real-type*. The required real-type-identifier **real** shall denote the real-type. The values shall be an implementation-defined subset of the real numbers denoted as specified in 6.1.5 by signed-real.

(c) *Boolean-type*. The required ordinal-type-identifier **Boolean** shall denote the Boolean-type. The values shall be the enumeration of truth values denoted by the required constant-identifiers **false** and **true**, such that **false** is the predecessor of **true**. The ordinal numbers of the truth values denoted by **false** and **true** shall be the integer values 0 and 1 respectively.

(d) *char-type*. The required ordinal-type-identifier **char** shall denote the char-type. The values shall be the enumeration of a set of implementation-defined characters, some possibly without graphic representations. The ordinal numbers of the character values shall be values of integer-type, that are implementation-defined, and that are determined by mapping the character values on to consecutive non-negative integer values starting at zero. The mapping shall be order preserving. The following relations shall hold.

- (1) The subset of character values representing the digits 0 to 9 shall be numerically ordered and contiguous.
- (2) The subset of character values representing the upper-case letters A to Z, if available, shall be alphabetically ordered but not necessarily contiguous.
- (3) The subset of character values representing the lower-case letters a to z, if available, shall be alphabetically ordered but not necessarily contiguous.
- (4) The ordering relationship between any two character values shall be the same as between their ordinal numbers.

NOTE: Operators applicable to the required simple-types are specified in 6.7.2.

6.4.2.3. Enumerated-Types. An enumerated-type shall determine an ordered set of values by enumeration of the identifiers that denote those values. The ordering of these values shall be determined by the sequence in which their identifiers are enumerated, i.e. if *x* precedes *y* then *x* is less than *y*. The ordinal number of a value that is of an enumerated-type shall be determined by mapping all the values of the type on to consecutive

non-negative values of integer-type starting from zero. The mapping shall be order preserving.

enumerated-type = "(" identifier-list ")" .

identifier-list = identifier { "," identifier } .

The occurrence of an identifier *in* the identifier-list *of* an enumerated-type shall constitute its defining-point as a constant-identifier for the region that is the block *closest-containing* the enumerated-type.

EXAMPLES:

(red, yellow, green, blue, tartan)
(club, diamond, heart, spade)
(married, divorced, widowed, single)
(scanning, found, notpresent)
(Busy, InterruptEnable, ParityError, OutOfPaper, LineBreak)

6.4.2.4. Subrange-Types. A subrange-type shall include identification of the smallest and the largest value in the subrange. The first constant *of* a subrange-type shall specify the smallest value, and this shall be less than or equal to the largest value which shall be specified by the other constant *of* the subrange-type. Both constants shall be of the same ordinal-type, and that ordinal-type shall be designated the host type of the subrange-type.

subrange-type = constant ".." constant .

EXAMPLES:

1..100
-10..+10
red..green
'0'..'9'

6.4.3. Structured-Types.

6.4.3.1. General. A new-structured-type shall be classified as an array-type, record-type, set-type or file-type according to the unpacked-structured-type *closest-contained* by the new-structured-type. A component of a value of a structured-type shall be a value.

structured-type = new-structured-type | structured-type-identifier .

new-structured-type = ["packed"] unpacked-structured-type .

unpacked-structured-type = array-type | record-type |
set-type | file-type .

The occurrence of the token **packed** *in* a new-structured-type shall designate the type denoted thereby as *packed*. The designation of a structured-type as packed shall indicate to the processor that data-storage of values should be economized, even if this causes operations on, or accesses to components of, variables possessing the type to be less efficient in terms of space or time.

The designation of a structured-type as packed shall affect the representation in data-storage of that structured-type only; i.e., if a component is itself structured, the component's representation in data-storage shall be packed only if the type of the component is designated packed.

NOTE: The ways in which the treatment of entities of a type is affected by whether or not the type is designated packed are specified in 6.4.3.2, 6.4.5, 6.6.3.3, 6.6.5.4 and 6.7.1.

6.4.3.2. Array-Types. An array-type shall be structured as a mapping from each value specified by its index-type on to a distinct component. Each component shall have the type denoted by the type-denoter *of* the component-type *of* the array-type.

array-type =
"array" "[" index-type { "," index-type } "]" "of" component-type .

index-type = ordinal-type .

component-type = type-denoter .

EXAMPLES:

array [1..100] of real
array [Boolean] of colour

An array-type that specifies a sequence of two or more index-types shall be an abbreviated notation for an array-type specified to have as its index-type the first index-type in the sequence, and to have a component-type that is an array-type specifying the sequence of index-types without the first and specifying the same component-type as the original specification. The component-type thus constructed shall be designated packed if and only if the original array-type is designated packed. The abbreviated form and the full form shall be equivalent.

NOTE: Each of the following two examples thus contains different ways of expressing its array-type.

EXAMPLE 1.

array [Boolean] of array [1..10] of array [size] of real
array [Boolean] of array [1..10, size] of real
array [Boolean, 1..10, size] of real
array [Boolean, 1..10] of array [size] of real

EXAMPLE 2.

packed array [1..10, 1..8] of Boolean
packed array [1..10] of packed array [1..8] of Boolean

Let i denote a value of the index-type; let $v[i]$ denote a value of that component of the array-type that corresponds to the value i by the structure of the array-type; let the smallest and largest values specified by the index-type be denoted by m and n ; and let $k = (\text{ord}(n) - \text{ord}(m) + 1)$ denote the number of values specified by the index-type; then the values of the array-type shall be the distinct k -tuples of the form

$(v[m], \dots, v[n])$.

NOTE: A value of an array-type does not therefore exist unless all of its component values are defined. If the component-type has c values, then it follows that the cardinality of the set of values of the array-type is c raised to the power k .

Any type designated packed and denoted by an array-type having as its index-type a denotation of a subrange-type specifying a smallest value of 1 and a largest value of greater than 1, and having as its component-type a denotation of the char-type, shall be designated a *string-type*.