PASCAL USERS GROUP

PASCAL NEWS

NUMBER 18

COMMUNICATIONS ABOUT THE PROGRAMMING LANGUAGE PASCAL BY PASCALERS

MAY, 1980

7185

POLICY: PASCAL NEWS (17-Mar-80)

* Pascal News is the official but informal publication of the User's Group.

* Pascal News contains all we (the editors) know about Pascal; we use it as the vehicle to answer all inquiries because our physical energy and resources for answering individual requests are finite. As PUG grows, we unfortunately succumb to the reality of:

  1. Having to insist that people who need to know "about Pascal" join PUG and read Pascal News - that is why we spend time to produce it!

  2. Refusing to return phone calls or answer letters full of questions - we will pass the questions on to the readership of Pascal News. Please understand what the collective effect of individual inquiries has at the "concentrators" (our phones and mailboxes). We are trying honestly to say: "We cannot promise more that we can do."

* Pascal News is produced 3 or 4 times during an academic year; usually in September, November, February, and May.

* ALL THE NEWS THAT'S FIT, WE PRINT. Please send material (brevity is a virtue) for Pascal News single-spaced and camera-ready (use dark ribbon and 18.5 cm lines!)

* Remember: ALL LETTERS TO US WILL BE PRINTED UNLESS THEY CONTAIN A REQUEST TO THE CONTRARY.

* Pascal News is divided into flexible sections:

  POLICY - explains the way we do things (ALL-PURPOSE COUPON, etc.)

  EDITOR'S CONTRIBUTION - passes along the opinion and point of view of the editor together with changes in the mechanics of PUG operation, etc.

  HERE AND THERE WITH PASCAL - presents news from people, conference announcements and reports, new books and articles (including reviews), notices of Pascal in the news, history, membership rosters, etc.

  APPLICATIONS - presents and documents source programs written in Pascal for various algorithms, and software tools for a Pascal environment; news of significant applications programs. Also critiques regarding program/algorithm certification, performance, standards conformance, style, output convenience, and general design.

  ARTICLES - contains formal, submitted contributions (such as Pascal philosophy, use of Pascal as a teaching tool, use of Pascal at different computer installations, how to promote Pascal, etc.).

  OPEN FORUM FOR MEMBERS - contains short, informal correspondence among members which is of interest to the readership of Pascal News.

  IMPLEMENTATION NOTES - reports news of Pascal implementations: contacts for maintainers, implementors, distributors, and documentors of various implementations as well as where to send bug reports. Qualitative and quantitative descriptions and comparisons of various implementations are publicized. Sections contain information about Portable Pascals, Pascal Variants, Feature-Implementation Notes, and Machine-Dependent Implementations.

- - - - - - ALL-PURPOSE COUPON - - - - - - (17-Mar-80)

Pascal User's Group, c/o Rick Shaw
Digital Equipment Corporation
5775 Peachtree Dunwoody Road
Atlanta, Georgia 30342 USA

NOTE

- Membership is for an academic year (ending June 30th).
- Membership fee and All Purpose Coupon is sent to your Regional Representative.
- SEE THE POLICY SECTION ON THE REVERSE SIDE FOR PRICES AND ALTERNATE ADDRESS if you are located in the European or Australasian Regions.
- Membership and Renewal are the same price.
- The U. S. Postal Service does not forward Pascal News.

- -

[ ] Enter me as a new member for:
[ ] Renew my subscription for:

[ ] 1 year ending June 30, 1980
[ ] 2 years ending June 30, 1981
[ ] 3 years ending June 30, 1982

[ ] Send Back Issue(s) _____

[ ] My new/correct address/phone is listed below

[ ] Enclosed please find a contribution, idea, article or opinion which is submitted for publication in the Pascal News.

[ ] Comments: _____

_____

_____

ENCLOSED PLEASE FIND: $ / A$ / £ ____.____

NAME _____

ADDRESS _____

_____

_____

PHONE _____

COMPUTER _____

DATE _____

JOINING PASCAL USER'S GROUP?

- Membership is open to anyone: Particularly the Pascal user, teacher, maintainer, implementor, distributor, or just plain fan.
- Please enclose the proper prepayment (check payable to "Pascal User's Group"); we will not bill you.
- Please do not send us purchase orders; we cannot endure the paper work!
- When you join PUG any time within an academic year: July 1 to June 30, you will receive all issues of Pascal News for that year.
- We produce Pascal News as a means toward the end of promoting Pascal and communicating news of events surrounding Pascal to persons interested in Pascal. We are simply interested in the news ourselves and prefer to share it through Pascal News. We desire to minimize paperwork, because we have other work to do.

- American Region (North and South America): Send $6.00 per year to the address on the reverse side. International telephone: 1-404-252-2600.
- European Region (Europe, North Africa, Western and Central Asia): Join through PUG (UK). Send £4.00 per year to: Pascal Users Group, c/o Computer Studies Group, Mathematics Department, The University, Southampton SO9 5NH, United Kingdom; or pay by direct transfer into our Post Giro account (28 513 4000); International telephone: 44-703-559122 x700.
- Australasian Region (Australia, East Asia - incl. Japan): PUG(AUS). Send $A8.00 per year to: Pascal Users Group, c/o Arthur Sale, Department of Information Science, University of Tasmania, Box 252C GPO, Hobart, Tasmania 7001, Australia. International telephone: 61-02-23 0561 x435

PUG(USA) produces Pascal News and keeps all mailing addresses on a common list. Regional representatives collect memberships from their regions as a service, and they reprint and distribute Pascal News using a proof copy and mailing labels sent from PUG(USA). Persons in the Australasian and European Regions must join through their regional representatives. People in other places can join through PUG(USA).

RENEWING?

- Please renew early (before August) and please write us a line or two to tell us what you are doing with Pascal, and tell us what you think of PUG and Pascal News. Renewing for more than one year saves us time.

ORDERING BACK ISSUES OR EXTRA ISSUES?

- Our unusual policy of automatically sending all issues of Pascal News to anyone who joins within a academic year (July 1 to June 30) means that we eliminate many requests for backissues ahead of time, and we don't have to reprint important information in every issue--especially about Pascal implementations!
- Issues 1 .. 8 (January, 1974 - May 1977) are out of print.
  (A few copies of issue 8 remain at PUG(UK) available for £2 each.)
- Issues 9 .. 12 (September, 1977 - June, 1978) are available from PUG(USA) all for $10.00 and from PUG(AUS) all for $A10.
- Issues 13 .. 16 are available from PUG(UK) all for £6; from PUG(AUS) all for $A10; and from PUG(USA) all for $10.00.
- Extra single copies of new issues (current academic year) are: $3.00 each - PUG(USA); £2 each - PUG(UK); and $A3 each - PUG(AUS).

SENDING MATERIAL FOR PUBLICATION?

- Your experiences with Pascal (teaching and otherwise), ideas, letters, opinions, notices, news, articles, conference announcements, reports, implementation information, applications, etc. are welcome. Please send material single-spaced and in camera-ready (use a dark ribbon and lines 18.5 cm wide) form.
- All letters will be printed unless they contain a request to the contrary.

Contributors to this issue (#18) were:

EDITOR	Rick Shaw
Here & There	John Eisenberg
Books & Articles	Rich Stevens
Applications	Rich Cichelli, Andy Mickel
Standards	Jim Miner, Tony Addyman
Implementation Notes	Bob Dietrich
Administration	Moe Ford, Kathy Ford, Jennie Sinclair

APPLICATION FOR LICENSE TO USE VALIDATION SUITE FOR PASCAL

Name and address of requestor:
(Company name if requestor is a company) _____

Phone Number: _____

Name and address to which information should be addressed (Write "as above" if the same) _____

Signature of requestor: _____

Date: _____

In making this application, which should be signed by a responsible person in the case of a company, the requestor agrees that:

a) The Validation Suite is recognized as being the copyrighted, proprietary property of R. A. Freak and A.H.J. Sale, and

b) The requestor will not distribute or otherwise make available machine-readable copies of the Validation Suite, modified or unmodified, to any third party without written permission of the copyright holders.

In return, the copyright holders grant full permission to use the programs and documentation contained in the Validation Suite for the purpose of compiler validation, acceptance tests, benchmarking, preparation of comparative reports, and similar purposes, and to make available the listings of the results of compilation and execution of the programs to third parties in the course of the above activities. In such documents, reference shall be made to the original copyright notice and its source.

X Distribution charge: $50.00

X Make checks payable to ANPA/RI in US dollars drawn on a US bank. Remittance must accompany application.

Mail request to:

ANPA/RI
P.O. Box 598
Easton, Pa. 18042
USA
Attn: R.J. Cichelli

Source Code Delivery Medium Specification:
9-track, 800 bpi, NRZI, Odd Parity, 600' Magnetic Tape

( ) ANSI-Standard

a) Select character code set:
( ) ASCII ( ) EBCDIC

b) Each logical record is an 80 character card image.
Select block size in logical records per block.
( ) 40 ( ) 20 ( ) 10

( ) Special DEC System Alternates:
( ) RSX-IAS PIP Format
( ) DOS-RSTS FLX Format

Office use only

Signed _____
Date _____

Richard J. Cichelli
On behalf of A.H.J. Sale & R.A. Freak

Editor's Contribution

Wow! Bet ya didn't expect to see another edition of Pascal News so soon! Actually, PN #17 was so late that we printed both editions at the same time.

ABOUT THIS ISSUE

In an effort to keep our members up to date with activity on the standards front, we have devoted this whole issue to the proposed ISO draft standard.

It is very important that our members review this proposal and comment if they feel it necessary. The national standards body in your country, or a member of the standards committee is the best person to send any comments. (See also Tony Addyman's comments on returning comments.)

ON BEING ON TIME

As you have probably noticed, Pascal News is still not back on a proper schedule. So, what are we doing about it? Well I'll tell you. We are working very hard. Honest! The plan is to publish PN #19 in June. This would get us almost up to date for this year. Then work through the summer to get PN #20 out by September. This would make us almost on schedule, right? The reason that we can be so optimistic is that all the set up work for an operation in Atlanta (as opposed to Minneapolis) has been completed. Now all we have to do is crank out the news!

THE BAD NEWS

Inflation has hit PUG. As of 1-July-80 the membership fee for Pascal Users Group will have to be raised. It will not be much, but at least enough to cover the cost of printing and mailing. We are loosing money every issue now. In the U.S. at the moment it is only a few cents a copy. But at $1.43 a copy for returned issues by the Post Office we are getting killed. Note that you members can help with this problem, by always informing us of your new address when you move.

THANKS

We all owe Tony Addyman a debt of gratitude, for the years (literally!) of work that has gone into the proposal for an ISO standard for the language Pascal. Without his drive and enthusiasm, the standard for Pascal would still be just a good idea.

A Draft Propasal for Pascal

A.M.Addyman
Dept of Computer Science
University of Manchester
Oxford Road
Manchester, M13 9PL, United Kingdom

CONTENTS

0. FOREWORD TO THE DRAFT

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

(a) 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.

(b) to define a language whose implementations could be both reliable and efficient on then available computers.

However, it has become apparent that Pascal has attributes which go far beyond these original goals. It is now being increasingly used commercially in the writing of both 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.

1. SCOPE OF THIS STANDARD

1.1 This Standard specifies requirements for
(a) the syntax of Pascal;
(b) the semantic rules for interpreting the meaning of a program written in Pascal;
(c) the form of input data to be processed by a program written in Pascal;
(d) the form of output data produced by a program written in Pascal.

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

2. REFERENCES

None.

3. DEFINITIONS

(a) error. A violation by a program of the requirements of this standard.
(b) implementation-defined. Those parts of the language which may differ between processors, but which will be defined for any particular processor.
(c) implementation-dependent. Those parts of the language which may differ between processors, and for which there need not be a definition for a particular processor.
(d) processor. A compiler, interpreter, or other mechanism which accepts the program as input and either executes it, prepares it for execution, or both.
(e) scope. The text for which the declaration or definition of an identifier or label is valid.
(f) totally-undefined. If a variable is of a structured-type, the state of the variable when every component of the variable is totally-undefined. Totally-undefined is synonymous with undefined if the variable is not of a structured-type.

(g) undefined. The state of a variable or function when the variable or function does not have attributed to it a value of its type.

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 meta-symbols. Further specification of the constructs is given by prose and, in some cases, by equivalent program fragments. Any identifier that is defined in clause 6 as the identifier of a predeclared or predefined entity shall denote that 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

Meta-symbol	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|...|z)	grouping: any one of x,y,...,z
"xyz"	the terminal symbol xyz
lower-case-name	a non-terminal symbol

For increased readability, the non-terminal symbols are hyphenated. 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 may 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 are those implicitly required to form the tokens and separators defined in 6.1.

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 clause 6 with the meanings defined in clause 6;
(b) be accompanied by a document that provides a definition of all implementation-defined features;
(c) treat each occurrence of 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 have reported a prior warning 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.
The method for reporting errors or warnings shall be implementation-dependent.
(d) be accompanied by a document that separately describes any features accepted by the processor that are not specified in clause 6. Such extensions shall be described as being 'extensions to Pascal specified by ISO....: 198-'.
(e) be able to process in a manner similar to that specified for errors any use of any such extension;
(f) be able to process in a manner similar to that specified for errors any use of an implementation-dependent feature.

5.2 Programs
A program complying with the requirements of this standard shall:
(a) use only those features of the language specified in clause 6;
(b) not rely on any particular interpretation of implementation-dependent features.

6. REQUIREMENTS

6.1 Lexical tokens

NOTE. The syntax given in this sub-clause (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 case of any letter 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 shall serve to denote constants, types, variables, procedures, functions parameters, bounds and programs, and fields and tag-fields in records. Identifiers may be of any length. No identifier shall have the same spelling as any word-symbol.

```
identifier = letter {(letter | digit)} .
```

Examples:

```
X          time       readinteger  sum    AlterHeatSetting
InquireWorkstationTransformation
InquireWorkstationIdentification
```

6.1.4 Directives. Directives shall only occur as a replacement for a procedure-block or function-block. The directive forward shall be the only standard directive (see 6.6.1 and 6.6.2). Other implementation-dependent directives may be defined. No directive shall have the same spelling as any word-symbol.

```
directive = letter {(letter | digit)} .
```

6.1.5 Numbers. Decimal notation shall be used for numbers that are the constants of integer-type and real-type (see 6.4.2.2). The letter "e" preceding a scale factor shall mean 'times ten to the power of'. The value of an unsigned-integer shall be in the closed interval 0 to maxint (see 6.4.2.2).

```
digit-sequence = digit {digit} .
unsigned-integer = digit-sequence .
unsigned-real =
    unsigned-integer "." digit-sequence ["e" scale-factor] |
    unsigned-integer "e" scale-factor .
unsigned-number = unsigned-integer | unsigned-real .
scale-factor = signed-integer .
sign = "+" | "-" .
signed-integer = [sign] unsigned-integer .
signed-real = [sign] unsigned-real .
signed-number = signed-integer | signed-real .
```

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, that shall be in the closed interval 0 to 9999.

```
label = digit-sequence .
```

6.1.7 Character-strings. A character-string consisting of a single string-element shall denote a constant of char-type (see 6.4.2.2). A character-string consisting of enclosed string-elements shall denote a constant of a string-type (see 6.4.3.2) with the same number of components as the character-string has string-elements. If the string of characters is to contain an apostrophe, this apostrophe shall be denoted by an apostrophe-image. Each string-character shall denote an implementation-defined value of char-type.

```
character-string = "'" string-element
    {string-element} "'" .
string-element = apostrophe-image | string-character .
apostrophe-image = "''" .
string-character =
    one-of-an-implementation-defined-set-of-characters .
```

Examples:

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

6.1.8 Token separators. The construct

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

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

Comments, spaces (except in character-strings), and ends of 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, or unsigned-numbers. No separators shall occur within tokens.

6.2 Blocks and scope

6.2.1 Block. A block shall consist of the definitions, declarations and statement-part that together form a part of a procedure-declaration, of a function-declaration or of a program.

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

The label-declaration-part shall specify all labels that prefix a statement in the corresponding statement-part. Each declared label shall prefix at most one statement in the statement-part. The

occurrence of a label as part of a label-declaration-part shall be its defining-point for the region that is the block immediately containing the label-declaration-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 .
```

All variables whose identifiers are declared in the variable-declaration-part of a block, except for those listed as program-parameters, shall be totally-undefined when execution of the statement-part of their block commences.

6.2.2 Scope

6.2.2.1 Each identifier or label within the block of a Pascal program 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, 6.6.1, 6.6.2, 6.6.3.1, 6.8.3.10).

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 that has a defining-point for region A has a further 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 field-identifier of a field-designator (see 6.5.3.3) shall be one of the field-identifiers associated with the type of the record-variable.

6.2.2.7 The scope of a defining-point of an identifier or label shall include no other defining-point of the same identifier or label.

6.2.2.8 Within the scope of a defining-point of an identifier or label, all other occurrences of that identifier or label shall be designated corresponding occurrences. No occurrence outside that scope shall be a corresponding occurrence.

6.2.2.9 A defining-point of an identifier or label shall precede all corresponding occurrences of that identifier or label in the program-block with one exception, namely that a type-identifier T, that denotes the domain of a pointer-type ↑T, may have its defining-point anywhere within the type-definition-part in which ↑T occurs.

6.2.2.10 Identifiers that denote standard constants, types, procedures and functions shall be used as if their defining-points have a region enclosing the program.

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

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

```
constant-definition = identifier "=" constant .
constant = [sign] (unsigned-number ¦ constant-identifier)
         ¦ character-string .
constant-identifier = identifier .
```

The occurrence of an identifier as the left-hand side of a constant-definition shall be its defining-point, at the end of the constant-definition, for the region that is the block immediately containing the constant-definition-part in which the constant-definition occurs. Each corresponding occurrence of that identifier shall be a constant-identifier and shall denote the constant of the constant-definition. A constant-identifier preceded by a sign shall have been defined to denote a value of real-type or of integer-type.

6.4 Type-definitions

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

```
type-definition = identifier "=" type-denoter .
type-denoter = type-identifier ¦ new-type .
new-type = simple-type ¦ structured-type ¦ pointer-type .
```

The occurrence of an identifier as the left-hand side of a type-definition shall be its defining-point, at the end of the type-definition, for the region that is the block immediately containing the type-definition-part in which the type-definition occurs. Each corresponding occurrence of that identifier shall be a type-identifier and shall denote the same type as is denoted by its type-denoter.

Types shall be classified as simple, structured or pointer types according to the new-type with which they have been denoted. There shall be in addition certain predefined types which shall be denoted by predefined type-identifiers (see 6.4.2.2 and 6.4.3.5). 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.

```
simple-type-identifier = type-identifier .
structured-type-identifier = type-identifier .
pointer-type-identifier = type-identifier .
type-identifier = identifier .
```

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.

```
simple-type = ordinal-type | real-type .
ordinal-type = enumerated-type | subrange-type |
               integer-type | Boolean-type | char-type |
               ordinal-type-identifier .
```

Where an appropriate word is substituted for x, an x-type-identifier shall be a type-identifier defined to denote an x-type.

6.4.2.2 Standard simple-types. The following types shall be standard:

integer-type — The predefined integer-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 the signed-integer values (see also 6.7.2.2). The ordinal number of a value of integer-type shall be the value itself.

real-type — The predefined 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 the signed-real values.

Boolean-type — The predefined Boolean-type-identifier Boolean shall denote the Boolean-type. The values shall be the enumeration of truth values denoted by the predefined 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.

char-type — The predefined char-type-identifier char shall denote the char-type. The type 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:

(a) The subset of character values representing the digits 0 to 9 shall be numerically ordered and

contiguous.

(b) The subset of character values representing the upper-case letters A to Z, if available, shall be alphabetically ordered but not necessarily contiguous.

(c) The subset of character values representing the lower-case letters a to z, if available, shall be alphabetically ordered but not necessarily contiguous.

(d) The ordering relationship between any two character values shall be the same as between their ordinal numbers.

NOTE. Operators applicable to standard 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 as they occur in the identifier-list of the enumerated-type on to consecutive non-negative integer values starting from zero.

```
enumerated-type = "(" identifier-list ")" .
identifier-list = identifier { "," identifier } .
```

The occurrence of an identifier as part of the identifier-list of an enumerated-type shall be its defining-point as a constant-identifier for the region that is the block immediately containing the type-definition-part or variable-declaration-part in which the enumerated-type occurs.

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. The definition of a type as a subrange of an ordinal-type shall include identification of the smallest and the largest value in the subrange. The first constant shall specify the smallest value which shall be less than or equal to the largest value. 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. Structured-types shall be classified as array, record, set or file types according to the unpacked-structured-type immediately contained in their denotation. A component of a value of a structured-type shall be a value.

```
structured-type = ["packed"] unpacked-structured-type |
                  structured-type-identifier .
unpacked-structured-type = array-type | record-type | set-type |
                           file-type .
```

A structured-type which immediately contains an unpacked-structured-type shall be designated packed if and only if the token packed is immediately contained in the structured-type. The designation of a structured-type as packed shall indicate to the processor that data-storage should be economised, even if this causes operations on, or accesses to components of, variables of the type to be less efficient in terms of space or time.

The designation of a structured-type as packed shall affect only the representation in data-storage of that structured-type. 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. Sections 6.4.3.2, 6.4.5, 6.6.3.3, and 6.6.5.4 specify the ways in which the treatment of entities of a type is affected by whether or not the type is designated packed.

6.4.3.2 Array-types. An array-type shall be structured as a mapping from each value of its index-type onto a distinct component. The index-type shall be an ordinal-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 alternative notation for an array-type specified to have the index-type of 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.

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 of the index-type be denoted by m and n; and let k = (ord(n)-ord(m)+1) denote the number of values of the index-type. Then the values of the array-type shall be the distinct k-tuples of the form:
```
(v[m], ... ,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 denoted by

```
packed array[T1] of T2
```

where T1 is a subrange-type with a lower bound of 1 and T2 is the char-type, shall be designated a string-type.

NOTE. The values of a string-type possess additional properties which determine their correspondence with character-strings (see 6.1.7), allow writing them to textfiles (see 6.9.4.7) and define their use with relational-operations (see 6.7.2.5).

6.4.3.3 Record-types. A record-type shall be structured as a fixed number of components that shall be designated fields.

The occurrence of an identifier as a tag-field or as part of the identifier-list of a record-section shall be its defining-point as a field-identifier for the region that is the record-type immediately containing the tag-field or record-section. Each field-identifier shall be associated with a component of the specified type.

Let a variant-part contained in a field-list be considered as an additional field with appropriate values, and let Vi denote a value of the i-th field in a record-type definition with m fields. Then the record-type shall have a single null value if it has no fields; otherwise it shall have only the set of values:

V1, ... ,Vm

NOTE. If the number of values in each of the fields is F1,F2,...,Fm; then it follows that the cardinality of the set of values of the record-type is (F1*F2*...*Fm).

If the record-type contains a variant-part, the tag-type of that variant-part shall be an ordinal-type. All the case-constants of that variant-part shall be distinct and shall be of a type compatible with the tag-type (see 6.4.5). The set of values of all the case-constants shall be equal to the set of values of the tag-type.

Let each field-list immediately contained in a variant of a variant-part be considered to be a record-type with values as defined above. Then, if the variant-part contains a tag-field in its variant-selector or if its variants immediately contain no case-constant-lists with more than one case-constant, the variant-part shall have only the values:

k,Xk

where k denotes a value in the tag-type and Xk denotes a value of the variant associated with k. The occurrence of a case-constant in the case-constant-list of a variant shall associate the value of the case-constant with that variant.

NOTE. If there are n values in the tag-type, and the variant associated with the value i has Ti values, then it follows that the cardinality of the set of values of the variant-part is (T1+T2+...+Tn).

If a variant-part contains no tag-field in its variant-selector and it immediately contains case-constant-lists with more than one case-constant, then its values shall be determined as follows. Let f(i) denote an implicit function mapping values of the tag-type onto a new ordinal-type that shall have as many values as there are variants in the variant-part, and let the mapping be determined by associating with each variant in turn one value of this new type that is the result of applying f to each of the values of the tag-type in the case-constant-list associated with that variant. Then this case shall be equivalent to the one given before with the substitution of this new type for the tag-type and appropriate substitution of the case-constant-lists.

NOTE. A record-value exists only when none of its fields are undefined. A value of a variant-part exists when one and only one of its variants has a value.

The value of a tag-field shall determine which variant is active in determining the value of a variant-part. It shall be an error if any field-identifier defined within a variant is used in a field-designator (see 6.5.3.3) unless the value of the tag-field is associated with that variant. A variant-part that does not contain a tag-field in its variant-selector shall be assumed to have a virtual tag-field of the constructed ordinal-type described above and a

reference to a field of a variant shall attribute to the virtual tag-field the value of the constructed ordinal-type that is associated with that variant. Whenever a new variant is selected, the fields of that variant shall be totally-undefined unless they have been attributed a value subsequent to the change of variant.

```
record-type = "record" [field-list [";"]] "end" .
field-list = fixed-part [ ";" variant-part ] | variant-part .
fixed-part = record-section { ";" record-section } .
record-section = identifier-list ":" type-denoter .
variant-part = "case" variant-selector "of"
                  variant { ";" variant } .
variant-selector = [tag-field ":"] tag-type .
tag-field = identifier .
variant = case-constant-list ":"  "(" [ field-list [";"] ] ")" .
tag-type = ordinal-type-identifier .
case-constant-list = case-constant { "," case-constant } .
case-constant = constant .
```

Examples:

```
record
  year : 0..2000;
  month : 1..12;
  day : 1..31
end

record
  name, firstname : string;
  age : 0..99;
  case married : Boolean of
  true : (Spousesname : string);
  false: ()
end

record
  x,y : real;
  area : real;
  case shape of
  triangle :
    (side : real;
     inclination, angle1, angle2 : angle);
  rectangle :
    (side1, side2 : real;
     skew : angle);
  circle :
    (diameter : real);
end
```

6.4.3.4 Set-types. A set-type shall determine the set of values that is structured as the powerset of its base-type. Thus each value of a set-type shall be a set whose members shall be unique values of the base-type. If the base-type is the integer-type or a subrange thereof, the largest and smallest values of the base-type shall lie within limits which are implementation-defined.

set-type = "set" "of" base-type .
base-type = ordinal-type .

NOTE. Operators applicable to values of set-types are specified in 6.7.2.4.

6.4.3.5 File-types.

NOTE. A file-type describes sequences of values of the specified component-type, together with a current position in each sequence and a mode which indicates whether the sequence is being inspected or generated.

file-type = "file" "of" component-type .

A type-denoter shall not be permissible as the component-type of a file-type if it denotes either a file-type or a structured-type having any component whose type-denoter is not permissible as the component-type of a file-type.

A file-type shall define implicitly a type designated a sequence-type having exactly those values, which shall be designated sequences, defined by the following five rules.

(a) S() shall be a value of the sequence-type S, and shall be called the empty sequence. The empty sequence shall have no components.

(b) Let c be a value of the specified component-type, and let x be a value of the sequence-type S. Then S(c) shall be a sequence of S, consisting of the single component value c, and S(c)~x shall also be a sequence, distinct from S(), of type S.

(c) Let c, S, and x be as in (b); let y denote the sequence S(c)~x; and let z denote the sequence x~S(c); then the notation y.first shall denote c (i.e., the first component value of y), y.rest shall denote x (i.e., the sequence obtained from y by deleting the first component), and z.last shall denote c (i.e., the last component value of z).

(d) Let x and y each be a non-empty sequence of type S; then x = y shall be true if and only if both (x.first = y.first) and (x.rest = y.rest) are true. If x is the empty sequence, then x = y shall be true if and only if y is also the empty sequence.

(e) Let x, y, and z be sequences of type S; then x~(y~z) = (x~y)~z shall be true.

NOTE. The notation x~y represents the concatenation of sequences x and y. The explicit representation of sequences (e.g. S(c)), of concatenation of sequences, of the first, last and rest selectors, and of sequence equality is not part of the Pascal language. These notations are used to define file values, below, and the standard file operations in 6.6.5.2 and 6.6.6.5.

A file-type also shall define implicitly a type designated a

mode-type having exactly two values which are designated Inspection and Generation.

NOTE. The explicit denotation of these values is not defined in the Pascal language.

A file-type shall be structured as three components. Two of these components, designated f.L and f.R, shall be of the implicit sequence-type. The third component, designated f.M, shall be of the implicit mode-type.

Let f.L and f.R each be a single value of the sequence-type; let f.M be a single value of the mode-type; then each value of the file-type shall be a distinct three-tuple of the form

(f.L, f.R, f.M)

where f.R shall be only the empty sequence if f.M is the value Generation. The value, f, of the file-type shall be designated empty if and only if f.L~f.R is the empty sequence.

NOTE. The two components, f.L and f.R, of a value of the file-type may be considered to represent the single sequence f.L~f.R together with a current position in that sequence. If f.R is non-empty, then f.R.first may be considered the current component as determined by the current position; otherwise, the current position is called the end-of-file position.

A standard file-type shall be denoted by the predefined structured-type-identifier text. The component-type implicitly specified by type text shall be the standard type char. The structure of type text shall define an additional sequence-type whose values are designated lines. A line shall be a sequence x~S(e), where x is a sequence of components of type char, and e represents a special component value, which shall be designated an end-of-line, and which shall be indistinguishable from the char value space (denoted ' ') except by the standard function eoln (6.6.6.5) and by the standard procedures reset (6.6.5.2), writeln (6.9.5), and page (6.9.6). If x is a line then no component of x other than x.last shall be an end-of-line. This definition shall not be construed to determine the underlying representation, if any, of an end-of-line component used by a processor.

A line-sequence, z, shall be either the empty sequence or the sequence x~y where x is a line and y is a line-sequence.

Every value t of type text shall satisfy one of the following two rules.

(a) If t.M = Inspection, then t.L~t.R shall be a line-sequence.

(b) If t.M = Generation, then t.L~t.R shall be x~y where x is a line-sequence and y is a sequence of components of type char.

NOTE. In rule (b), y may be considered, especially if it is non-empty, to be a partial line which is being generated. Such a

partial line cannot occur during inspection of a file.

A variable declared to be of type text shall be called a textfile.

NOTE. All standard procedures and functions applicable to a variable of type file of char are applicable to textfiles. Additional standard procedures and functions, applicable only to textfiles, are defined in 6.6.6.5 and 6.9.

6.4.4 Pointer-types. The values of a pointer-type shall consist of a single nil-value, and a set of identifying-values each identifying a distinct variable of the domain-type. The set of identifying-values shall be dynamic, in that the variables and the values identifying them, may be created and destroyed during the execution of the program. Pointer values and the variables identified by them shall be created only by the standard procedure new (see 6.6.5.3).

NOTE. Since the nil-value is not an identifying-value it does not identify a variable.

The token nil shall denote the nil-value in all pointer-types.

pointer-type = "↑" domain-type | pointer-type-identifier .
domain-type = type-identifier .

NOTE. The token nil does not have a single type, but assumes a suitable type to satisfy the assignment-compatibility rules, or the compatibility rules for operators, if possible.

6.4.5 Compatible types. Types T1 and T2 shall be designated compatible if any of the four statements that follow is true.
(a) T1 and T2 are the same type.
(b) T1 is a subrange of T2, or T2 is a subrange of T1, or both T1 and T2 are subranges of the same host type.
(c) T1 and T2 are set-types of compatible base-types, and either both T1 and T2 are designated packed or neither T1 nor T2 is designated packed.
(d) T1 and T2 are string-types with the same number of components.

6.4.6 Assignment-compatibility. A value of type T2 shall be designated assignment-compatible with a type T1 if any of the five statements that follow is true.
(a) T1 and T2 are the same type, that is neither a file-type nor a structured-type with a file component (this rule is to be interpreted recursively).
(b) T1 is the real-type and T2 is the integer-type.
(c) T1 and T2 are compatible ordinal-types and the value of type T2 is in the closed interval specified by the type T1.
(d) T1 and T2 are compatible set-types and all the members of the value of type T2 are in the closed interval specified by the base-type of T1.
(e) T1 and T2 are compatible string-types.

At any place where the rule of assignment-compatibility is used:

(a) It shall be an error if T1 and T2 are compatible ordinal-types and the value of type T2 is not in the closed interval specified by the type T1.
(b) It shall be an error if T1 and T2 are compatible set-types and any member of the value of type T2 is not in the closed interval specified by the base-type of the type T1.

6.4.7 Example of a type-definition-part

```
type
  natural = 0..maxint;
  count = integer;
  range = integer;
  colour = (red, yellow, green, blue);
  sex = (male, female);
  year = 1900..1999;
  shape = (triangle, rectangle, circle);
  punchedcard = array[1..80] of char;
  string = file of char;
  polar = record
            r : real;
            theta : angle
          end;
  indextype = 1..limit;
  vector = array [indextype] of real;
  person = ↑persondetails;
  persondetails =
          record
            name, firstname : string;
            age : integer;
            married : Boolean;
            father, child, sibling : person;
            case s : sex of
              male :
                (enlisted,bearded : Boolean);
              female :
                (mother,programmer : Boolean)
          end;
  tape = file of person;
  FileOfInteger = file of integer;
```

NOTE. In the above example count, range and integer denote the same type. The types denoted by year and natural are compatible with, but not the same as, the type denoted by range, count and integer.

NOTE. Types occurring in examples in the remainder of this standard should be assumed to have been declared as specified in 6.4.7.

6.5 Declarations and denotations of variables

6.5.1 Variable-declarations. A variable is an entity to which a (current) value may be attributed (see 6.8.2.2). A variable-declaration shall consist of a list of identifiers denoting the distinct variables, followed by a denotation of their type.

```
variable-declaration = identifier-list ":" type-denoter .
```

The occurrence of an identifier as part of the identifier-list of a variable-declaration shall be its defining-point as a variable-identifier of the given type for the region that is the block immediately containing the variable-declaration-part in which the variable-declaration occurs. A variable declared in the variable-declaration-part of a block shall exist from the time the block is activated, until its statement-part is completed.

NOTE. This implies that each activation of a block introduces a distinct set of variables.

The structure of a variable of a structured-type shall be the structure of the structured-type.

Example of a variable-declaration-part

```
var
  x,y,z,max: real;
  i,j: integer;
  k: 0..9;
  p,q,r: Boolean;
  operator: (plus, minus, times);
  a: array[0..63] of real;
  c: colour;
  f: file of char;
  hue1,hue2: set of colour;
  p1,p2: person;
  m,m1,m2 : array[1..10,1..10] of real;
  coord : polar;
  pooltape : array[1..4] of FileOfInteger;
  date : record month : 1..12; year : integer end;
```

A variable-access, according to whether it is an entire-variable, a component-variable a referenced-variable or a buffer-variable, denotes either a declared variable, or a component of a variable, a variable which is identified by a pointer value (see 6.4.4) or a buffer-variable.

```
variable-access = entire-variable ¦ component-variable ¦
                  referenced-variable ¦ buffer-variable .
```

NOTE. Variables occurring in examples in the remainder of this standard should be assumed to have been declared as specified in 6.5.1.

6.5.2 Entire-variables. The identifier of an entire-variable denotes the variable of the corresponding variable-declaration, value-parameter-specification or variable-parameter-specification (see 6.6.3.1).

```
entire-variable = variable-identifier .
variable-identifier = identifier .
```

6.5.3 Component-variables

6.5.3.1 General. A component of a variable of a structured-type shall be a variable and shall be denoted by a component-variable. The type of a component-variable shall be the type of the specified component. The value, if any, of the component of a variable shall be the same component of the value, if any, of the variable.

```
component-variable = indexed-variable | field-designator .
```

6.5.3.2 Indexed-variables. A component of a variable of an array-type shall be denoted by an indexed-variable.

```
indexed-variable =
       array-variable "[" index-expression
       { "," index-expression } "]" .
array-variable = variable-access .
index-expression = expression .
```

The action of selecting a particular component of an array-variable shall be designated indexing. An array-variable shall be a variable of an array-type. The value of each index expression shall be assignment-compatible with the corresponding index-type specified in the definition of the array-type. The component denoted by the indexed-variable shall be the component that corresponds to the value of the index-expression by the mapping of the type of the array-variable (see 6.4.3.2).

Examples:

```
a[12]
a[i+j]
```

If the array-variable is itself an indexed-variable an abbreviation may be used. In the abbreviated form, all the index expressions shall be contained within the same enclosing square-brackets, a single comma replacing the sequence of right-square-bracket left-square-bracket that occurred in the full form. The abbreviated form shall be equivalent to the full form.

Examples:

```
m[k][1]
m[k,1]
```

NOTE. The two examples denote the same component variable.

6.5.3.3 Field-designators. A field-designator shall denote the component of the record-variable that is associated with the field-identifier by the type of the record-variable (see 6.2.2.6 and 6.4.3.3). A record-variable shall be a variable of a record-type.

```
field-designator = record-variable "." field-identifier .
record-variable = variable-access .
field-identifier = identifier .
```

Examples:
p2↑.mother
coord.theta

6.5.4 Referenced-variables.
A referenced-variable shall denote the variable (if any) identified by the value of the pointer-variable (see 6.4.4 and 6.6.5.3).

```
referenced-variable = pointer-variable "↑" .
pointer-variable = variable-access .
```

A variable allocated by the standard procedure new (see 6.6.5.3) shall be accessible until it is deallocated by the standard procedure dispose (see 6.6.5.3) or until program execution terminates. A pointer-variable shall be a variable of a pointer-type. The use of a pointer-variable in a referenced-variable shall be designated de-referencing.
It shall be an error if the pointer-variable has a nil-value or is undefined at the time it is de-referenced.

Examples:
p1↑
p1↑.father↑
p1↑.sibling↑.father↑

6.5.5 Buffer-variables. A file-variable shall denote a variable of a file-type. With each file-variable shall be associated a variable of the component-type specified by the file-type, denoted by a buffer-variable containing the file-variable.

```
buffer-variable = file-variable "↑" .
file-variable = variable .
```

Examples:
input↑
pooltape[2]↑

It shall be an error if the value of a file-variable f is altered while the buffer-variable is an actual variable parameter, or an element of the record-variable-list of a with-statement, or both. It shall be an error if the value of a file-variable f is altered by an assignment-statement which contains the buffer-variable f↑ in its left-hand side.

6.6 Procedure and function declarations
6.6.1 Procedure-declarations. A procedure-declaration shall associate an identifier with a procedure-block so that it can be activated by a procedure-statement. Activation of the procedure shall activate the procedure-block.

```
procedure-declaration =
        procedure-heading ";" directive |
        procedure-identification ";" procedure-block |
        procedure-heading ";" procedure-block .
```

```
  procedure-heading =
          "procedure" identifier [ formal-parameter-list ]  .
  procedure-identification =
          "procedure" procedure-identifier .
  procedure-identifier = identifier .
  procedure-block = block .
```

The procedure-heading shall specify the identifier denoting the procedure and the formal parameters (if any).

The occurrence of an identifier as part of the procedure-heading of a procedure-declaration shall be its defining-point as a procedure-identifier for the region that is the block immediately containing the procedure-and-function-declaration-part in which the procedure-declaration occurs.

The defining-point of a procedure-identifier shall be followed by a declaration of its procedure-block. Where the procedure-heading and the procedure-block occur in separate procedure-declarations, the correspondence shall be established by the use of the procedure-identifier that denoted the procedure.

In the case where the declaration of the procedure-block immediately follows the declaration of the procedure-heading, an abbreviation shall be allowed where the sequence

```
  directive ";" "procedure" procedure-identifier ";"
```

may be omitted between the procedure-heading and the procedure-block. The abbreviation shall be equivalent to the full notation.

Examples:

```
procedure readinteger (var f: text; var x: integer);
var
    i:natural;
begin
    while f↑ = ' ' do get(f);
    {The file buffer contains the first non-space char}
    i := 0;
    while f↑ in ['0'..'9'] do begin
        i := (10 * i) + (ord(f↑) - ord('0'));
        get(f)
    end;
    {The file buffer contains a non-digit}
    x := i
    {Of course if there are no digits, x is zero}
end;

procedure AddVectors(var A,B,C: array[low..high: natural] of real);
var
    i : natural;
begin
    for i := low to high do A[i] := B[i] + C[i]
end { of AddVectors };
```

```
procedure bisect(function f(x : real) : real;
                 a,b: real;
                 var result: real);
{This procedure attempts to find a zero of f(x) in (a,b) by
 the method of bisection. It is assumed that the procedure is
 called with suitable values of a and b such that
        (f(a)<0) and (f(b)>0)
 The estimate is returned in the last parameter.}
var
    midpoint: real;
begin
    {The invariant P is true by calling assumption}
    while abs(a-b) > 1e-10*abs(a) do begin
        midpoint := (a+b)/2;
        if f(midpoint) < 0 then a := midpoint
        else b :=midpoint
        {Which re-establishes the invariant:
            P = (f(a)<0) and (f(b)>0)
         and reduces the interval (a,b) provided that the value
         of midpoint is distinct from both a and b.}
    end;
    {P together with the loop exit condition assures that a zero
     is contained in a small sub-interval. Return the midpoint as
     the zero.}
    result := midpoint
 end;

procedure ConditionForAppending(var f: FileOfInteger);
{This procedure takes a file in an arbitrary state and sets
 it up in a condition for appending data to its end. Simpler
 conditioning is only possible if assumptions are made about the
 initial state of the file.}
var
    LocalCopy : FileOfInteger;

    procedure CopyFiles(var from,to : FileOfInteger);
    begin
        reset(from); rewrite(to);
        while not eof(from) do begin
            to↑ := from↑;
            put(to); get(from)
        end;
    end { of CopyFiles };

begin {of body of ConditionForAppending}
 CopyFiles(f,LocalCopy);
 CopyFiles(LocalCopy,f)
end { of ConditionForAppending };
```

6.6.2 Function-declarations. Function-declarations shall associate an identifier with a function-block so that it can be activated by a function-designator. Activation of the function shall activate the function-block.

```
function-declaration =
        function-heading ";" directive |
        function-identification ";" function-block |
        function-heading ";" function-block .
function-heading =
        "function" identifier [[formal-parameter-list]
        ":" result-type]  .
function-identification =
        "function" function-identifier .
function-identifier = identifier .
result-type = simple-type-identifier |
              pointer-type-identifier .
function-block = block .
```

The function-heading shall specify the identifier denoting the function, the formal parameters (if any), and the type of the function result. The function-block shall contain at least one assignment-statement that attributes a value to the function-identifier (see 6.8.2.2). The value of the function shall be the last value attributed to the function-identifier. If no assignment occurs during the activation of the function-block the function shall be undefined.

The occurrence of an identifier as part of the function-heading of a function-declaration shall be its defining-point as a function-identifier of the type denoted by the result-type for the region that is the block immediately containing the procedure-and-function-declaration-part in which the function-declaration occurs.

The defining-point of a function-identifier shall be followed by a declaration of its function-block. Where the function-heading and the function-block occur in separate function-declarations, the correspondence shall be established by the use of the function-identifier that denoted the function.

In the case where the declaration of the function-block immediately follows the declaration of the function-heading, an abbreviation shall be allowed where the sequence

```
directive ";" "function" function-identifier ";"
```

may be omitted between the function-heading and the function-block. The abbreviation shall be equivalent to the full notation.

Examples:

```
function Sqrt(x:real): real;
{This function computes the square root of x (x>0)
 using Newton's method.}
var
    old,new: real;
begin
    new := x;
    repeat
        old := new;
        new := (old + x/old)*0.5;
    until abs(new-old) < Eps*new;
    {Eps being a global constant}
    Sqrt := new
end { of Sqrt };

function GCD(m,n: natural): natural;
forward;

function max(a: vector; n: indextype): real;
{This function finds the largest value in a, which is declared
       a: array[indextype] of real
 and where
       indextype = 1..limit}
var
    largestsofar: real;
    fence: indextype;
begin
    largestsofar := a[1];
    {Establishes largestsofar = max(a[1])}
    for fence := 2 to limit do begin
        if largestsofar < a[fence] then largestsofar := a[fence]
        {Re-establishing largestsofar = max(a[1], ... ,a[fence])}
    end;
    {So now largestsofar = max(a[1], ... ,a[limit])}
    max := largestsofar
 end { of max };


function GCD;
{Parameters omitted as this completes a forward declaration}
begin
    if n=0 then GCD := m else GCD := GCD(n,m mod n);
end;
```

6.6.3 Parameters

6.6.3.1 General. There shall be four kinds of parameters : value parameters, variable parameters, procedural parameters and functional parameters. An identifier-list in a value-parameter-specification shall be a list of value parameters. An identifier-list in a variable-parameter-specification shall be a list of variable parameters.

```
formal-parameter-list =
       "(" formal-parameter-section
       {";" formal-parameter-section} ")" .
formal-parameter-section =
       value-parameter-specification |
       variable-parameter-specification |
       procedural-parameter-specification |
       functional-parameter-specification .
value-parameter-specification =
       identifier-list ":" type-identifier .
variable-parameter-specification =
       "var" identifier-list ":"
       (type-identifier | conformant-array-schema) .
conformant-array-schema =
       "array" "[" index-type-specification
       { ";" index-type-specification } "]" "of"
       ( type-identifier | conformant-array-schema ) .
index-type-specification =
       identifier ".." identifier
       ":" ordinal-type-identifier .
bound-identifier = identifier .
procedural-parameter-specification =
       procedure-heading .
functional-parameter-specification =
       function-heading .
```

An identifier that is defined to be a parameter-identifier in a formal-parameter-list shall be designated a formal parameter of the corresponding function-block or procedure-block, if any.

The occurrence of an identifier as part of an identifier-list of a value-parameter-specification or a variable-parameter-specification shall be its defining-point as a parameter-identifier for the region that is the formal-parameter-list immediately containing it and its defining-point as a variable-identifier for the region that is the corresponding procedure-block or function-block, if any.

The occurrence of an identifier as part of an index-type-specification shall be its defining-point as a bound-identifier for the region that is the formal-parameter-list immediately containing it and for the region that is the corresponding procedure-block or function-block, if any.

The occurrence of an identifier as part of a procedure-heading in a procedural-parameter-specification shall be its defining-point as a parameter-identifier for the region that is the formal-parameter-list immediately containing it and its defining-point as a procedure-identifier for the region that is the corresponging procedure-block or function-block, if any.

The occurrence of an identifier as part of a function-heading in a functional-parameter-specification shall be its defining-point as a parameter-identifier for the region that is the formal-parameter-list immediately containing it and its defining-point as a function-identifier for the region that is the

corresponding procedure-block or function-block, if any.

NOTE. If the formal-parameter-list is within a procedural-parameter-specification or a functional-parameter-specification, there is no corresponding procedure-block or function-block.

If the component of a conformant-array-schema is itself a conformant-array-schema, then an abbreviated form of definition may be used. In the abbreviated form, all the index-type-specifications shall be contained within the same enclosing square brackets, a single semi-colon replacing each sequence of right-square-bracket "of" "array" left-square-bracket that occurred in the full form. The abbreviated form shall be equivalent to the full form.

Examples:
```
array[u..v: T1] of array[j..k: T2] of T3
array[u..v: T1; j..k: T2] of T3
```

6.6.3.2 Value parameters. The formal parameter shall denote a distinct variable of the specified type. The actual-parameter (see 6.7.3 and 6.8.2.3) shall be an expression whose value is assignment-compatible with the type of the formal parameter. The current value of the expression shall be attributed upon activation of the block to the variable that is denoted by the formal parameter.

6.6.3.3 Variable parameters. The actual-parameter shall be a variable-access. The actual-parameters (see 6.7.3 and 6.8.2.3) corresponding to formal parameters that occur in a variable-parameter-specification shall all be of the same type. This type shall be the same as the type denoted by the type-identifier in the variable-parameter-specification if the formal parameter is so specified, otherwise it shall be conformable to the conformant-array-schema in the variable-parameter-specification. Each formal parameter shall denote the corresponding actual-parameter during the entire activation of the block.

If access to the actual-parameter involves the indexing of an array and/or a reference to a field within a variant of a record and/or the de-referencing of a pointer-variable and/or a reference to a buffer-variable, these actions shall be executed before the activation of the block.

Components of variables of any type designated packed shall not be used as actual variable parameters.

If T1 is an array-type, and T2 is the type denoted by the ordinal-type-identifier of the index-type-specification of a conformant-array-schema, then T1 is conformable with the conformant-array-schema if all the following four statements are true.

(a) The index-type of T1 is compatible with T2.
(b) The smallest and largest value of the index-type of T1 lie within the closed interval defined by values of T2.

(c) The component-type of T1 is the same as the component type of the conformant-array-schema, or is conformable to the component conformant-array-schema.
(d) T1 is not designated packed.

It shall be an error if the smallest or largest value of the index-type of T1 lies outside the closed interval defined by the values of T2.

During the entire activation of the block, the first bound-identifier shall denote the smallest value of the index-type of the actual-parameters, and the second bound-identifier shall denote the largest value of the index-type of the actual-parameters.

6.6.3.4 Procedural parameters. The actual-parameter (see 6.7.3 and 6.8.2.3) shall be a procedure-identifier that has a defining-point in the program-block. The actual procedure and the formal procedure shall have congruous formal-parameter-lists (see 6.6.3.6). The formal parameter shall denote the actual parameter during the entire activation of the block. If the procedural parameter, upon activation, accesses any entity whose region encloses the procedure-block, the entity accessed shall be one that would have been accessible to the procedure-declaration when its procedure-identifier was passed as a procedural parameter.

6.6.3.5 Functional parameters. The actual-parameter (see 6.7.3 and 6.8.2.3) shall be a function-identifier that has a defining-point in the program-block. The actual function and the formal function shall have congruous formal-parameter-lists (see 6.6.3.6) and the same result-type. The formal parameter shall denote the actual parameter during the entire activation of the block. If the functional parameter, upon activation, accesses any entity whose region encloses the function-block, the entity accessed shall be one that would have been accessible to the function-declaration when its function-identifier was passed as a functional parameter.

6.6.3.6 Parameter list congruity. Two formal-parameter-lists shall be congruous if they contain the same number of formal-parameter-sections and if the formal-parameter-sections in corresponding positions match. Two formal-parameter-sections shall match if any of the four statements that follow is true.
(a) They are both value-parameter-specifications containing the same number of parameters that are of the same type.
(b) They are both variable-parameter-specifications containing the same number of parameters that are of the same type, or have equivalent conformant-array-schemas. Two conformant-array-schemas are equivalent if they have the same ordinal-type specified in their index-type-specifications and their components are either of the same type or are equivalent conformant-array-schemas.
(c) They are both procedural-parameter-specifications with congruous parameter lists, if any.
(d) They are both functional-parameter-specifications with congruous parameter lists, if any, and the same result-type.

6.6.4 Standard procedures and functions

6.6.4.1 General. Standard procedures and functions shall be predeclared. The standard procedures and functions shall be as specified in 6.6.5 and 6.6.6 respectively.

6.6.5 Standard procedures

6.6.5.1 General. The standard procedures shall be file handling procedures, dynamic allocation procedures and transfer procedures.

6.6.5.2 File handling procedures. The effects of applying each of the file handling procedures rewrite, put, reset and get to a file-variable f shall be defined by pre-assertions and post-assertions about f, its components f.L, f.R, and f.M, and about the associated buffer-variable f↑. The use of the variable g within an assertion shall be considered to represent the state or value, as appropriate, of f prior to the operation, and similarly for g↑ and f↑, while f (within an assertion) shall denote the variable after the operation.

It shall be an error if the stated pre-assertion does not hold immediately prior to any use of the defined operation. It shall be an error if any variable explicitly denoted in an assertion of equality is undefined. The post-assertion shall hold prior to the next subsequent reference to the file, its components, or the buffer- variable.

```
rewrite(f)  pre-assertion:  true.
            post-assertion: (f.L = f.R = S()) and
                            (f.M = Generation) and
                            (f↑ is totally-undefined).

put(f)      pre-assertion:  (g.M = Generation) and
                            (g.L is not undefined) and
                            (g.R = S()) and
                            (g↑ is not undefined).
            post-assertion: (f.M = Generation) and
                            (f.L = (g.L~S(g↑))) and
                            (f.R = S()) and
                            (f↑ is totally undefined).

reset(f)    pre-assertion:  The components g.L and g.R are not
                            undefined.
            post-assertion: (f.L = S()) and
                            (f.R = (g.L~g.R~X)) and
                            (f.M = Inspection) and
                            (if f.R = S() then (f↑ is
                            totally-undefined)
                            else (f↑ = f.R.first)),

                            where, if f is of type text and if
                            g.L~g.R is not empty and if
                            (g.L~g.R).last is not designated an
                            end-of-line, then X shall be a sequence
                            having an end-of-line component as its
                            only component; otherwise X = S().
```

get(f) pre-assertion: (g.M = Inspection) and
(neither g.L nor g.R are undefined) and
(g.R <> S()).
post-assertion: (f.M = Inspection) and
(f.L = (g.L~g.R.first)) and
(f.R = g.R.rest) and
(if f.R = S() then (f↑ is
totally-undefined)
else (f↑ = f.R.first)).

When the file-variable f is of a type other than text, the standard procedures read and write shall be defined as follows.

read Read(f,v1,...,vn) where v1...vn denote variables shall be equivalent to

begin read(f,v1); ... ; read(f,vn) end

Read(f,v) where v denotes a variable shall be equivalent to

begin v := f↑; get(f) end

write Write(f,e1,...,en), where e1...en denote expressions shall be equivalent to

begin write(f,e1); ... ; write(f,en) end

Write(f,e), where e denotes an expression shall be equivalent to

begin f↑ := e; put(f) end

NOTE. The standard procedures read, write, readln, writeln, and page, as applied to textfiles, are described in 6.9.

6.6.5.3 Dynamic allocation procedures
The parameters p and q shall be variable-parameters of a pointer-type.

new(p) shall allocate a new variable v that is totally-undefined. The pointer-variable p shall have attributed to it a value that identifies the new variable v.

new(p,c1,...,cn) shall allocate a new variable v that is totally-undefined. The pointer-variable p shall have attributed to it a value that identifies the new variable v. The allocated variable shall have nested variants that correspond to the case-constants c1,...,cn. The case-constants shall be listed in order of increasing nesting of the variant-parts. Any variant not specified shall be at a deeper level of nesting than that specified by cn. It shall be an error to change any variant-part of the allocated variable from a variant specified.

dispose(q) shall indicate that the variable q↑ is no longer accessible. All pointers that referenced this variable shall become undefined. It shall be an error if the variable q↑ had been allocated using the form new(p,c1,...,cn).

dispose(q,k1,...,km) shall indicate that the variable q↑, whose variants correspond to the case-constants k1,...,km, is no longer accessible. The case-constants shall be listed in order of increasing nesting of the variant-parts. All pointers that referenced this variable shall become undefined. It shall be an error if the variable had been allocated using the form new(p,c1,...,cn) and m is less than n. It shall be an error if the variants in the variable q↑ are different from those specified by the case-constants k1...km.

It shall be an error if the pointer parameter of dispose has a nil-value or is undefined.

It shall be an error if a variable that is identified by the pointer parameter of dispose (or a component thereof) is currently either an actual variable parameter, or an element of the record-variable-list of a with-statement, or both.

It shall be an error if a referenced-variable created using the second form of new is used in its entirety as an operand in an expression, or as the variable in an assignment-statement or as an actual-parameter.

6.6.5.4 Transfer procedures

Let a be a variable of a type denoted by 'array [s1] of T', let z be a variable of a type denoted by 'packed array [s2] of T' , and u and v be the smallest and largest values of the type s2, then the statement pack(a,i,z) shall be equivalent to

```
begin
k := i;
for j := u to v do
    begin
    z[j] := a[k];
    if j <> v then k := succ(k)
    end
end
```

and the statement unpack(z,a,i) shall be equivalent to

```
begin
k := i;
for j := u to v do
    begin
    a[k] := z[j];
    if j <> v then k := succ(k)
    end
end
```

where j and k denote auxiliary variables that do not occur elsewhere in the program. The type of j shall be s2, the type of k shall be s1, and i shall denote an expression of a type that is compatible with s1.

6.6.6 Standard functions

6.6.6.1 General. The standard functions shall be arithmetic functions, transfer functions, ordinal functions and Boolean functions.

6.6.6.2 Arithmetic functions. For the following arithmetic functions, the expression x shall be either of real-type or integer-type. For the functions abs and sqr, the type of the result shall be the same as the type of the parameter, x. For the remaining arithmetic functions, the result shall always be of real-type. It shall be an error if the mathematically defined result as defined below would fall outside the set of values of the indicated result type.

abs(x) shall compute the absolute value of x.
sqr(x) shall compute the square of x.
sin(x) shall compute the sine of x, where x is in radians.
cos(x) shall compute the cosine of x, where x is in radians.
exp(x) shall compute the value of the base of natural logarithms raised to the power x.
ln(x) shall compute the natural logarithm of x, if x is greater than zero. It shall be an error if x is not greater than zero.
sqrt(x) shall compute the non-negative square root of x, if x is not negative. It shall be an error if x is negative.
arctan(x) shall compute the principal value, in radians, of the arctangent of x.

6.6.6.3 Transfer functions

trunc(x) From the expression x that shall be of real-type, this function shall return a result of integer-type. The value of trunc(x) shall be such that if x is positive or zero then 0<=x-trunc(x)<1; otherwise -1<x-trunc(x)<=0. It shall be an error if such a value does not exist.
Examples:
trunc(3.7) yields 3
trunc(-3.7) yields -3

round(x) From the expression x that shall be of real-type, this function shall return a result of integer-type. If x is positive or zero, round(x) shall be equivalent to

trunc(x+0.5), otherwise round(x) shall be equivalent to trunc(x-0.5).
It shall be an error if such a value does not exist.
Examples:
round(3.7) yields 4
round(-3.7) yields -4

6.6.6.4 Ordinal functions

ord(x) The parameter x shall be an expression of an ordinal-type. The result that is of integer-type shall be the ordinal number (see 6.4.2.2 and 6.4.2.3) of the value of the expression x. It shall be an error if such a value does not exist.

chr(x) The parameter x shall be an expression of integer-type. The result shall be the value of char-type whose ordinal number is equal to the value of the expression x if such a character value exists. It shall be an error if such a character value does not exist.

NOTE. For any value, ch, of char-type, the following is true:

```
chr(ord(ch)) = ch
```

succ(x) The parameter x shall be an expression of an ordinal-type. The result shall be of the same type as that of the expression (see 6.7.1). The function shall yield a value whose ordinal number is one greater than that of the expression x, if such a value exists. It shall be an error if such a value does not exist.

pred(x) The parameter x shall be an expression of an ordinal-type. The result shall be of the same type as that of the expression (see 6.7.1). The function shall yield a value whose ordinal number is one less than that of the expression x, if such a value exists. It shall be an error if such a value does not exist.

6.6.6.5 Boolean functions

odd(x) The parameter x shall be an expression of integer-type. The function yields the value of the expression 'abs(x) mod 2 = 1'.

eof(f) The parameter f shall be a file-variable; if the actual-parameter-list is omitted, the function shall be applied to the standard textfile input (see 6.10). When eof(f) is activated, it shall be an error if f is undefined; otherwise the function shall yield the value true if f.R is the empty sequence (see 6.4.3.5), otherwise false.

eoln(f) The parameter f shall be a textfile; if the actual-parameter-list is omitted, the function shall be applied to the standard textfile input (see 6.10). When eoln(f) is activated, it shall be an error if f is undefined or if eof(f) is true; otherwise the function shall yield the value true if f.R.first is an end-of-line component (see 6.4.3.5), otherwise false.

6.7 Expressions

6.7.1 General. An expression shall possess a value unless a variable or function-designator used as a factor in that expression is undefined at the time of its use, in which case an error shall occur.

The use of a variable-access as a factor shall denote the value, if any, attributed to the variable denoted by that variable-access. Operator precedences shall be according to four classes of operators as follows. The operator not shall have the highest precedence, followed by the multiplying-operators, then the adding-operators and signs, and finally, with the lowest precedence, the relational-operators. Sequences of two or more operators of the same precedence shall be left associative.

```
unsigned-constant = unsigned-number | character-string |
                    constant-identifier | "nil" .
factor = variable-access | unsigned-constant | bound-identifier |
         function-designator | set-constructor |
         "(" expression ")" | "not" factor .
set-constructor = "[" [ member-designator
         { "," member-designator } ] "]" .
member-designator = expression [ ".." expression ] .
term = factor { multiplying-operator factor } .
simple-expression = [ sign ] term { adding-operator term } .
expression =
    simple-expression [ relational-operator simple-expression ] .
```

Any factor whose type is S, where S is a subrange of T, shall be treated as of type T. Similarly, any factor whose type is set of S shall be treated as of an anonymous type set of T, and any factor whose type is packed set of S shall be treated as of an anonymous type packed set of T.

> NOTE. Consequently an expression that consists of a single factor of type S shall itself be of type T, and an expression that consists of a single factor of type set of S shall itself be of type set of T, and an expression that consists of a single factor of type packed set of S shall itself be of type packed set of T.

A set-constructor shall denote a value of a set-type. The set-constructor [] shall denote that value in every set-type that contains no members. A set-constructor containing one or more member-designators shall denote a value of type set of T or packed set of T, where T is the type of all expressions immediately contained in all member-designators of the set-constructor. The type T shall be an ordinal-type. The value denoted by the set shall contain zero or more members each of which shall be denoted by at least one member-designator of the set. It shall be an error if the value of any member denoted by any member-designator of the set-constructor is outside the implementation-defined limits (see 6.4.3.4).

The member-designator x, where x is an expression, shall denote the member that shall have the value x. The member-designator x..y,

where x and y are expressions, shall denote zero or more members that shall have the values of the base-type in the closed interval from the value of x to the value of y.

NOTE. The member-designator x..y denotes no members if the value of x is greater than the value of y.

A set-constructor shall assume a suitable type to satisfy the assignment-compatibility rules, or the compatibility rules for operators, if possible. It shall be an error if the possible types of a set-constructor do not permit it to assume a suitable type.

Examples are as follows:

(a) Factors:
```
x
15
(x+y+z)
sin(x+y)
[red,c,green]
[1,5,10..19,23]
not p
```

(b) Terms:
```
x*y
i/(1-i)
(x <= y) and (y < z)
```

(c) Simple expressions:
```
x+y
-x
hue1 + hue2
i*j + 1
```

(d) Expressions:
```
x = 1.5
p <= q
p = q and r
(i < j) = (j < k)
c in hue1
```

6.7.2 Operators
6.7.2.1 General

```
multiplying-operator = "*" | "/" | "div" | "mod" | "and" .

adding-operator = "+" | "-" | "or" .

relational-operator =
       "=" | "<>" | "<" | ">" | "<=" | ">=" | "in" .
```

A factor, or a term, or a simple-expression shall be designated an operand. The order of evaluation of the operands of a dyadic operator shall be implementation-dependent.

NOTE. This means, for example, that the operands may be evaluated in textual order, or in reverse order, or in parallel or they may not both be evaluated.

6.7.2.2 Arithmetic operators. The types of operands and results for dyadic and monadic operations shall be as shown in tables 2 and 3 respectively.

Table 2. Dyadic arithmetic operations

operator	operation	type of operands	type of result
+	addition	integer-type or real-type	)integer-type)if both
-	subtraction	integer-type or real-type	)operands are)of integer-type
*	multiplication	integer-type or real-type	)otherwise)real-type
/	division	integer-type or real-type	real-type
div	division with truncation	integer-type	integer-type
mod	modulo	integer-type	integer-type

Table 3. Monadic arithmetic operations

operator	operation	type of operand	type of result
+	identity	integer-type	integer-type
		real-type	real-type
-	sign-inversion	integer-type	integer-type
		real-type	real-type

NOTE. The symbols +, - and * are also used as set operators (see 6.7.2.4).

It shall be an error if j is zero, otherwise the value of i div j shall be such that
abs(i) - abs(j) < abs((i div j) * j) <= abs(i)
where the value shall be zero if abs(i)<abs(j), otherwise the sign of the value shall be positive if i and j have the same sign and negative if i and j have different signs.

It shall be an error if j is zero or negative, otherwise the value of i mod j shall be that value of (i-(k*j)) for integral k such that 0 <= i mod j < j.

The predefined constant maxint shall be of integer-type and shall denote an implementation-defined value. This value shall satisfy the following conditions:

(a) All integral values in the closed interval from -maxint to +maxint shall be values in the integer-type.
(b) Any monadic operation performed on an integer value in this interval shall be correctly performed according to the mathematical rules for integer arithmetic.
(c) Any dyadic integer operation on two integer values in this same interval shall be correctly performed according to the mathematical rules for integer arithmetic, provided that the result is also in this interval.
(d) Any relational operation on two integer values in this same interval shall be correctly performed according to the mathematical rules for integer arithmetic.

It shall be an error if the operation is not performed according to the mathematical rules for integer arithmetic.

6.7.2.3 Boolean operators. The types of operands and results for Boolean operations shall be as shown in table 4.

Table 4. Boolean operations

operator	operation	type of operand(s)	type of result
or	logical or	Boolean-type	Boolean-type
and	logical and	Boolean-type	Boolean-type
not	logical negation	Boolean-type	Boolean-type

Boolean-expression = expression .

A Boolean-expression shall be an expression that possesses a value of Boolean-type.

6.7.2.4 Set operators. The types of operands and results for set operations shall be as shown in table 5.

Table 5. Set operations

operator	operation	type of operands	type of result
+	set union	)	)
		)	)
-	set difference	)any set-type T	)T
		)	)
*	set intersection	)	)

6.7.2.5 Relational operators. The types of operands and results for relational operations shall be as shown in table 6.

Table 6. Relational operations

operator	type of operands	type of result
= <>	any set, simple, pointer or string-type	Boolean-type
< >	any simple or string-type	Boolean-type
<= >=	any set, simple or string-type	Boolean-type
in	left operand:any ordinal type T right operand: set of T(see 6.7.1)	Boolean-type

The operands of =, <>, <, >, >=, and <= shall be either of compatible type, or one operand shall be of real-type and the other shall be of integer-type.

The operators =, <>, <, > shall stand for "equal to", "not equal to", "less than" and "greater than" respectively.

Except when applied to sets, the operators <= and >= shall stand for "less than or equal to" and "greater than or equal to" respectively.

If u and v are operands of a set-type, u <= v shall denote the inclusion of u in v and u >= v shall denote the inclusion of v in u.

NOTE. Since the Boolean-type is an ordinal-type with false less than true, then if p and q are operands of Boolean-type, p = q denotes their equivalence and p <= q means p implies q.

When the relational operators = , <> , < , > , <= , >= are used to compare operands of a string-type (see 6.4.3.2), they denote lexicographic ordering according to the ordering of the character set (see 6.4.2.2).
The operator in shall yield the value true if the value of the operand of ordinal-type is a member of the value of the set, otherwise it shall yield the value false. In particular, if the value of the operand of ordinal-type is outside the implementation-defined limits on the base-type of the set (see 6.4.3.4), the operator in shall yield the value false.

6.7.3 Function designators. A function-designator shall yield the value of the function denoted by the function-identifier immediately contained within it. If the function has any formal parameters the function-designator shall contain a list of actual-parameters that shall be bound to their corresponding formal parameters defined in

the function-declaration. The correspondence shall be established by the positions of the parameters in the lists of actual and formal parameters respectively. The number of actual-parameters shall be equal to the number of formal parameters. The types of the actual-parameters shall correspond to the types of the formal parameters as specified by 6.6.3. The order of evaluation and binding of the actual-parameters shall be implementation-dependent.

```
function-designator = function-identifier
                        [ actual-parameter-list ] .
actual-parameter-list =
      "(" actual-parameter { "," actual-parameter } ")" .
actual-parameter = expression ¦ variable-access ¦
                   procedure-identifier ¦
                   function-identifier .
```

Examples:
```
Sum(a,63)
GCD(147,k)
sin(x+y)
eof(f)
ord(f↑)
```

6.8 Statements

6.8.1 General. Statements shall denote algorithmic actions, and shall be executable. They may be prefixed by a label.

If a label prefixes a simple-statement or structured-statement S, that label shall only be allowed in goto-statements (see 6.8.2.4) in the statement S, or in the statement-sequence (if any) in which S is immediatey contained and if this statement-sequence is the statement-sequence of the compound-statement that forms the statement-part of a block, the procedure-declarations and function-declarations of that block.

```
statement = [ label ":" ] ( simple-statement ¦
                            structured-statement ) .
```

NOTE. A goto-statement within a procedure may refer to a label in an enclosing procedure, provided that the label prefixes a simple-statement or structured-statement at the outermost level of nesting of the block of the procedure.

6.8.2 Simple-statements

6.8.2.1 General. A simple-statement shall be a statement of which no part constitutes another statement. An empty-statement shall consist of no symbols and shall denote no action.

```
simple-statement =
      empty-statement ¦ assignment-statement ¦
      procedure-statement ¦ goto-statement .
empty-statement = .
```

6.8.2.2 Assignment-statements. The assignment-statement shall attribute to the variable, denoted by the variable-access, or function-identifier a value, specified as an expression, that shall be assignment-compatible with the type of the variable or function.

An assignment-statement that has a function-identifier as its left-hand side shall occur only within the function-block (if any) that corresponds to the function denoted by the function-identifier.

```
assignment-statement =
    ( variable-access ¦ function-identifier ) ":=" expression .
```

If access to the variable involves the indexing of an array and/or a reference to a field within a variant of a record and/or the de-referencing of a pointer-variable and/or a reference to a buffer-variable, the decision whether these actions precede or follow the evaluation of the expression shall be implementation-dependent.

```
Examples:    x := y+z
             p := (1<=i) and (i<100)
             i := sqr(k) - (i*j)
             hue1 := [blue,succ(c)]
             p1↑.mother := true
```

6.8.2.3 Procedure-statements. A procedure-statement shall specify the activation of the procedure denoted by the procedure-identifier immediately contained within it. If the procedure has any formal parameters the procedure-statement shall contain a list of actual-parameters that shall be bound to their corresponding formal parameters defined in the procedure-declaration. The correspondence shall be established by the positions of the parameters in the lists of actual and formal parameters respectively. The number of actual-parameters shall be equal to the number of formal parameters. The types of the actual-parameters shall correspond to the types of the formal parameters as specified by 6.6.3. The order of evaluation and binding of the actual-parameters shall be implementation-dependent.

```
procedure-statement = procedure-identifier
                      [ actual-parameter-list ] .
```

```
Examples:    printheading
             transpose(a,n,m)
             bisect(fct,-1.0,+1.0,x)
```

6.8.2.4 Goto-statements. A goto-statement shall serve to indicate that further processing is to continue at another part of the program text, namely at the place prefixed by the label (see 6.8.1).

```
goto-statement = "goto" label .
```

6.8.3 Structured-statements

6.8.3.1 General. Structured-statements shall be constructs composed of other statements that have to be executed either in sequence (compound-statement), conditionally (conditional-statements), repeatedly (repetitive-statements), or within an expanded scope (with-statements).

```
structured-statement =
      compound-statement ¦ conditional-statement¦
      repetitive-statement ¦ with-statement .
```

6.8.3.2 Compound-statements. The compound-statement shall specify that its component statements are to be executed in textual order, except as modified by execution of a goto-statement.

```
compound-statement = "begin" statement-sequence "end" .
statement-sequence = statement { ";" statement } .
```

Example: begin z := x ; x := y; y := z end

6.8.3.3 Conditional-statements.

```
conditional-statement = if-statement ¦ case-statement .
```

6.8.3.4 If-statements

```
if-statement = "if" Boolean-expression "then" statement
               [ else-part ] .
else-part = "else" statement .
```

If the Boolean-expression yields the value true, the statement following the then shall be executed. If the Boolean-expression yields the value false, the action shall depend on the existence of an else-part; if the else-part is present the statement following the else shall be executed, otherwise an empty-statement shall be executed.

To resolve the so-called 'dangling-else' ambiguity, an if-statement without an else-part shall not be followed by the token else.

NOTE. An else-part thus becomes paired with the nearest preceding unpaired then.

Examples:

```
if x < 1.5 then z := x+y else z := 1.5
if p1 <> nil then p1 := p1↑.father
```

6.8.3.5 Case-statements. The case-statement shall consist of a case-index and a list of statements. Each statement shall be preceded by one or more case-constants. All the case-constants shall be distinct and shall be of the same ordinal-type as the case-index. The case-statement shall specify execution of the statement whose case-constant is equal to the value of the case-index upon entry to the case-statement.
It shall be an error if none of the case-constants is equal to the value of the case-index upon entry to the case-statement.

NOTE. Case-constants are not the same as statement labels.

```
case-statement =
       "case" case-index "of"
       case-list-element {";" case-list-element } [";"] "end" .
case-list-element = case-constant-list ":" statement .
case-index = expression .
```

Example:

```
case operator of
  plus:   x := x+y;
  minus:  x := x-y;
  times:  x := x*y
end
```

6.8.3.6. Repetitive-statements. Repetitive-statements shall specify that certain statements are to be executed repeatedly.

```
repetitive-statement = repeat-statement |
                       while-statement | for-statement .
```

6.8.3.7 Repeat-statements

```
repeat-statement = "repeat" statement-sequence
                   "until" Boolean-expression .
```

The sequence of statements between the tokens repeat and until shall be repeatedly executed (except as modified by the execution of a goto-statement) until the Boolean-expression yields the value true on completion of the statement-sequence. The statement-sequence shall be executed at least once, because the Boolean-expression is evaluated after execution of the statement-sequence.

Example:

```
repeat k := i mod j;
  i := j;
  j := k
until j = 0
```

6.8.3.8 While-statements

```
while-statement = "while" Boolean-expression "do" statement .
```

The while-statement

```
while b do body
```

shall be equivalent to

```
begin
if b then
  repeat
  body
  until not (b)
end
```

Examples:

```
while a[i] < x do i := i+1

while i>0 do
  begin if odd(i) then z := z*x;
    i := i div 2;
    x := sqr(x)
  end

while not eof(f) do
  begin process(f↑ ); get(f)
  end
```

6.8.3.9 For-statements. The for-statement shall specify that a statement is to be repeatedly executed while a progression of values is attributed to a variable that is designated the control-variable of the for-statement.

```
for-statement = "for" control-variable ":=" initial-value
                ( "to" ¦ "downto" ) final-value "do" statement .
control-variable = entire-variable .
initial-value = expression .
final-value = expression .
```

The control-variable shall be an entire-variable whose identifier is declared in the variable-declaration-part of the block immediately containing the for-statement. The control-variable shall be of an ordinal-type, and the initial-value and final-value shall be of a type compatible with this type.

An assigning-reference to a variable shall occur if any of the six statements that follow is true.

(a) The variable is denoted by a variable-access as the left hand side of an assignment-statement.
(b) The variable is denoted by an actual variable parameter in a function-designator or procedure-statement.
(c) The variable is denoted by an actual parameter in a procedure-statement that specifies the activation of the standard procedure read or the standard procedure readln.
(d) The variable occurs as the control-variable of a for-statement.
(e) A procedure-identifier in a procedure-statement or function-designator denotes a procedure-declaration that contains an assigning reference to the variable.
(f) A function-identifier in a procedure-stantment or function-designator denotes a function-declaration that contains an assigning reference to the variable.

Assigning references to the control-variable shall not occur within the repeated statement. It shall be an error if the final-value is not assignment-compatible with the control-variable when the initial-value is assigned to the control-variable. After a for-statement is executed (other than being left by a goto-statement leading out of it) the control-variable shall be undefined. Apart from the restrictions imposed by these requirements, the for-statement

```
for v := e1 to e2 do body
```

shall be equivalent to

```
begin
temp1 := e1;
temp2 := e2;
if temp1 <= temp2 then
  begin
  v := temp1;
  body;
  while v <> temp2 do
    begin
    v := succ(v);
    body
    end
  end
end
```

and the for-statement

```
for v := e1 downto e2 do body
```

shall be equivalent to

```
begin
temp1 := e1;
temp2 := e2;
if temp1 >= temp2 then
  begin
  v := temp1;
  body;
  while v <> temp2 do
    begin
    v := pred(v);
    body
    end
  end
end
```

where temp1 and temp2 denote auxiliary variables that do not occur elsewhere in the program and are of the type of the variable v if that type is not a subrange-type; otherwise of the host type of the variable v.

Examples:

```
for i := 2 to 63 do
  if a[i] > max then max := a[i]

for i := 1 to 10 do
  for j := 1 to 10 do
    begin
    x := 0;
    for k := 1 to 10 do
      x := x + m1[i,k]*m2[k,j];
    m[i,j] := x
    end

for c := red to blue do q(c)
```

6.8.3.10 With-statements

```
with-statement =
       "with" record-variable-list "do"
       statement .
record-variable-list =
       record-variable { "," record-variable } .
```

The occurrence of a record-variable as part of the record-variable-list shall be a defining-point of the field-identifiers of its record-type as variable-identifiers for the region that is a part of the with-statement immediately containing the record-variable-list. The region is that part of the with-statement that follows the record-variable. If access to a variable in the record-variable-list involves the indexing of an array and/or a reference to a field within a variant of a record and/or the de-referencing of a pointer-variable and/or a reference to a buffer-variable, these actions shall be executed before the component statement is executed.

The statement

```
with v1,v2, ...,vn do s
```

shall be equivalent to

```
with v1 do
  with v2 do
    ...
      with vn do s
```

Example:

```
with date do
if month = 12 then
  begin month := 1; year := year + 1
  end
else month := month+1
```

shall be equivalent to

```
if date.month = 12 then
  begin date.month := 1; date.year := date.year+1
  end
else date.month := date.month+1
```

6.9 Input and output

6.9.1 General. Textfiles (see 6.4.3.5) that are identified as program-parameters (see 6.10) to a Pascal program shall provide the standard legible input and output.

6.9.2 The procedure read. The syntax of the parameter list of read when applied to a textfile shall be:

```
read-parameter-list =
    "("[file-variable ","] variable-access
    {"," variable-access}")" .
```

If the file-variable is omitted, the procedure shall be applied to the standard textfile input.

The following requirements shall apply for the procedure read (where f denotes a textfile and v1...vn denote variables of char-type (or a subrange of char-type), integer-type (or a subrange of integer-type), or real-type):

(a) read(f,v1,...,vn) shall be equivalent to

```
begin read(f,v1); ... ; read(f,vn) end
```

(b) If v is a variable of char-type (or subrange thereof), read(f,v) shall be equivalent to

```
begin v := f↑; get(f) end
```

(c) If v is a variable of integer-type (or subrange thereof), read(f,v) shall cause the reading from f of a sequence of characters that form a signed-integer according to the syntax of 6.1.5. The value of the signed-integer thus read shall be assignment-compatible with the type of v, and shall be attributed to v. Preceding spaces and end-of-lines shall be skipped. Reading shall cease as soon as the file's buffer-variable f↑ contains a character that does not form part of the signed-integer. It shall be an error if the sequence of characters does not form a signed-integer as specified in 6.1.5.

(d) If v is a variable of real-type, read(f,v) shall cause the reading from f of a sequence of characters that form a signed-number according to the syntax of 6.1.5. The value denoted by the number thus read shall be attributed to the variable v. Preceding spaces and end-of-lines shall be skipped. Reading shall cease as soon as the file's buffer-variable f↑ contains a character that does not form part of the signed-number. It shall be an error if the sequence of

characters does not form a signed-number as specified in 6.1.5.

(e) When read is applied to f, it shall be an error if f is undefined or if f.M = Generation (see 6.4.3.5).

6.9.3 The procedure readln. The syntax of the parameter list of readln shall be:

```
readln-parameter-list =
    ["(" (file-variable | variable-access)
    {"," variable-access} ")"] .
```

Readln shall only be applied to textfiles. If the file-variable or readln-parameter-list is omitted, the procedure shall be applied to the standard textfile input.

readln(f,v1,...,vn) shall be equivalent to

```
begin read(f,v1,...,vn); readln(f) end
```

readln(f) shall be equivalent to

```
begin while not eoln(f) do get(f); get(f) end
```

NOTE. The effect or readln is to place the current file position just past the end of the current line in the textfile. Unless this is the end-of-file position, the current file position is therefore at the start of the next line.

6.9.4 The procedure write. The syntax of the parameter list of write when applied to a textfile shall be:

```
write-parameter-list =
    "("[file-variable ","] write-parameter
    {"," write-parameter}")" .
write-parameter =
    expression [":" expression [":" expression ] ] .
```

If the file-variable is omitted, the procedure shall be applied to the standard textfile output. When write is applied to a textfile, it shall be an error if f is undefined or f.M = Inspection (see 6.4.3.5).

6.9.4.1 Multiple parameters. Write(f,p1,...,pn) shall be equivalent to

```
begin write(f,p1); ... ; write(f,pn) end
```

where f denotes a textfile, and p1,...,pn denote write-parameters.

6.9.4.2 Write-parameters. The write-parameters p shall have the following forms:

```
e:TotalWidth:FracDigits     e:TotalWidth        e
```

where e is an expression whose value is to be written on the file f and may be of integer-type, real-type, char-type, Boolean-type or a string-type, and where TotalWidth and FracDigits are expressions of integer-type whose values are the field-width parameters. The values of TotalWidth and FracDigits shall be greater than or equal to one; it shall be an error if either value is less than one. Exactly TotalWidth characters shall be written (with an appropriate number of spaces to the left of the representation of e), except when e requires more than TotalWidth characters for its representation; in such a case the number of characters written shall be as small as is consistent with the representation of the value of e (see 6.9.4.4 and 6.9.4.5).

Write(f,e) shall be equivalent to the form write(f,e:TotalWidth), using a default value for TotalWidth that depends on the type of e; for integer-type, real-type and Boolean-type the default values shall be implementation-defined.

Write(f,e:TotalWidth:FracDigits) shall be applicable only if e is of real-type (see 6.9.4.5.2).

6.9.4.3 Char-type. If e is of char-type, the default value of TotalWidth shall be one. The representation written on the file f shall be:

(TotalWidth - 1) spaces,
the character value of e.

6.9.4.4 Integer-type. If e is of integer-type, the decimal representation of e shall be written on the file f. Assume a function

```
function IntegerSize ( x : integer ) : integer ;
   { returns the number of digits, z, such that
   10 to the power (z-1) <= abs(x) < 10 to the power z }
```

and let IntDigits be the positive integer defined by:

```
if e = 0
then IntDigits := 1
 else IntDigits := IntegerSize(e);
```

then the representation shall consist of:

(1) if TotalWidth >= IntDigits + 1 :
(TotalWidth - IntDigits - 1) spaces,
the sign character: '-' if e < 0, otherwise a space,
IntDigits characters of the decimal representation of abs(e).

(2) If TotalWidth < IntDigits + 1:
if e < 0 the sign character '-',
IntDigits characters of the decimal representation of abs(e).

6.9.4.5 Real-Type. If e is of real-type, a decimal representation of the number e, rounded to the specified number of significant figures or decimal places, shall be written on the file f.

6.9.4.5.1 The floating-point representation.
Write(f,e:TotalWidth) shall cause a floating-point representation of e to be written. Assume functions

```
function TenPower ( Int : integer ) : real ;
   { Returns 10.0 raised to the power Int }

function RealSize ( y : real ) : integer ;
   { Returns the value, z, such that
   TenPower(z-1) <= abs(y) < TenPower(z) }

   function Truncate ( y : real ; DecPlaces : integer )
   : real ;
   { Returns the value of y after truncation
   to DecPlaces decimal places }
```

let ExpDigits be an implementation-defined value representing the number of digit-characters written in an exponent;

let ActWidth be the positive integer defined by:

```
if TotalWidth >= ExpDigits + 6
   then ActWidth := TotalWidth
   else ActWidth := ExpDigits + 6;
```

and let the non-negative number eWritten and the integer ExpValue be defined by:

```
if e = 0.0
   then begin eWritten := 0.0; ExpValue := 0 end
   else
   begin
   eWritten := abs(e);
   ExpValue := RealSize ( eWritten ) - 1;
   eWritten := eWritten / TenPower ( ExpValue ) ;
   eWritten := eWritten +
      0.5*TenPower( - (ActWidth-ExpDigits-5) );
   if eWritten >= 10.0
      then
      begin
      eWritten := eWritten / 10.0;
      ExpValue := ExpValue + 1
      end;
   eWritten := Truncate ( eWritten, ActWidth
      - ExpDigits -5 )
   end;
```

then the floating-point representation of the value of e shall consist of:

```
the sign character,
   ( '-' if e < 0, otherwise a space )
the leading digit of the decimal representation of eWritten,
the character '.' ,
the next ( ActWidth - ExpDigits - 5 ) digits
   of the decimal representation of eWritten,
```

```
 an implementation-defined exponent character
    (either 'e' or 'E'),
 the sign of ExpValue
    ( '-' if ExpValue < 0, otherwise '+' ),
 the ExpDigits digits of the decimal representation of ExpValue
    (with leading zeros if the value requires them).
```

6.9.4.5.2 The fixed-point representation.
Write(f,e:TotalWidth:FracDigits) shall cause a fixed-point representation of e to be written. Assume the function IntegerSize described in clause 6.9.4.4, and the functions TenPower and Truncate described in clause 6.9.4.5.1;

let eWritten be the non-negative number defined by:

```
 if e = 0.0
    then eWritten := 0.0
    else
    begin
    eWritten := abs(e);
    eWritten := eWritten + 0.5
       * TenPower ( - FracDigits );
    eWritten := Truncate ( eWritten, FracDigits )
    end;
```

let IntDigits be the positive integer defined by:

```
 if trunc ( eWritten ) = 0
    then IntDigits := 1
    else IntDigits:= IntegerSize ( trunc(eWritten) );
```

and let MinNumChars be the positive integer defined by:

```
MinNumChars := IntDigits + FracDigits + 1;
 if e < 0.0
    then MinNumChars := MinNumChars + 1;{'-' required}
```

then the fixed-point representation of the value of e shall consist of:

```
 if TotalWidth >= MinNumChars,
    (TotalWidth - MinNumChars) spaces,
 the character '-' if e < 0,
 the first IntDigits characters of the decimal representation
    of the value of eWritten,
 the character '.',
 the next FracDigits characters of the decimal representation
    of the value of eWritten.
```

NOTE. At least MinNumChars characters are written. If TotalWidth is less than this value, no initial spaces are written.

6.9.4.6 Boolean-type. If e is of Boolean-type, a representation of the word true or the word false (as appropriate to the value of e) shall be written on the file f. This shall be equivalent to writing the appropriate character-strings 'True' or 'False' (see 6.9.4.7),

where the case of each letter is implementation-defined, with a field-width parameter of TotalWidth.

6.9.4.7 String-types. If the type of e is a string-type with n components, the default value of TotalWidth shall be n. The representation shall consist of:

```
if TotalWidth > n,
   (TotalWidth - n) spaces,
the characters e[1] through e[n] in that order.
```

6.9.5 The procedure writeln. The syntax of the parameter list of writeln shall be:

```
writeln-parameter-list =
    ["(" (file-variable | write-parameter)
    {"," write-parameter}")"] .
```

Writeln shall only be applied to textfiles. If the file-variable or the writeln-parameter-list is omitted, the procedure shall be applied to the standard textfile output.

writeln(f,p1,...,pn) shall be equivalent to

```
begin write(f,p1,...,pn);  writeln(f) end
```

Writeln shall be defined by a pre-assertion and a post-assertion using the notation of 6.6.5.2.

```
pre-assertion:  (g is not undefined) and (g.M = Generation).
post-assertion: (f.L = (g.L~S(e))) and
                (f.R = S()) and (f.M = Generation),
                where S(e) is the sequence consisting solely of the
                end-of-line component defined in 6.4.3.5.
```

NOTE. Writeln(f) terminates the partial line, if any, which is being generated. By the conventions of 6.6.5.2 it is an error if the pre-assertion is not true prior to the writeln(f).

6.9.6 The procedure page
It shall be an error if the pre-assertion required for writeln(f) (see 6.9.5) does not hold prior to a the application of page(f). If the actual-parameter-list is omitted the procedure shall be applied to the standard textfile output. Page(f) shall cause an implementation-defined effect on the textfile f, such that subsequent output to f will be on a new page if the textfile is printed on a suitable device, and shall perform an implicit writeln(f) if f.L.last is not the end-of-line component (see 6.4.3.5). The effect of inspecting a textfile to which the page procedure was applied during generation shall be implementation-dependent.

6.10 Programs. A Pascal program shall have the form of a procedure declaration except for its heading and its termination by a period.

```
program = program-heading ";" program-block "." .
```

```
program-heading =
      "program" identifier [ "(" program-parameters ")" ] .
program-parameters = identifier-list .
program-block = block .
```

The identifier following the token program shall be the program name which has no significance within the program. The program-parameters shall be distinct identifiers. The program-parameters shall be declared in the variable-declaration-part of the program-block. The binding of the variables denoted by the program-parameters to entities external to the program shall be implementation-dependent, except if the variable is of a file-type in which case the binding shall be implementation-defined.

> NOTE. The external representation of such external entities is not defined by this Standard, nor is any property of a Pascal program dependent on such representation. The appearance of an identifier in the program-parameters is not a defining-point nor a corresponding occurrence to a defining-point (see 6.2.2) because it is not in the program-block.

The two standard textfiles input and output shall not be declared explicitly, but shall be listed as parameters in the program-heading if they are used in the program-block. The occurrence of the identifiers input or output as program-parameters shall have the effect of declaring them as textfiles in the program block. The effects of the initialising statements reset(input) and rewrite(output) shall be caused to occur by the processor immediately following the begin of the block of the program if the respective identifier occurs in the program-parameters. The effect of an explicit use of reset or rewrite on the standard textfiles input or output shall be implementation-defined.

Examples:

```
program copy(f,g);
var f,g: file of real;
begin reset(f); rewrite(g);
      while not eof(f) do
         begin g↑ := f↑; get(f); put(g)
         end
end.
```

```
program copytext(input,output);
 {This  program  copies the characters and ends-of-lines of the
 textfile input to the textfile output.}
var ch: char;
begin
   while not eof do
   begin
      while not eoln do
         begin read(ch); write(ch)
         end;
      readln; writeln
   end
end.
```

```
program t6p6p3p3d2revised(output);
var globalone, globaltwo : integer;

procedure dummy;
  begin
  writeln('fail4...6.6.3.3-2')
  end { of dummy };

procedure p(procedure f(procedure ff; procedure gg);
           procedure g);
  var localtop : integer;
  procedure r;
     begin
     if globalone = 1 then
        begin
        if (globaltwo <> 2) or (localtop <> 1) then
           writeln('fail1...6.6.3.3-2')
        end
     else if globalone = 2 then
        begin
        if (globaltwo <> 2) or (localtop <> 2) then
           writeln('fail2...6.6.3.3-2')
        else
           writeln('pass...6.6.3.3-2')
        end
     else
        writeln('fail3...6.6.3.3-2');
     globalone := globalone + 1
     end { of r };
  begin { of p }
  globaltwo := globaltwo + 1;
  localtop := globaltwo;
  if globaltwo = 1 then
     p(f,r)
  else
     f(g,r)
  end { of p};
procedure q(procedure f; procedure g);
  begin
  f;
  g
  end;
begin
globalone := 1;
globaltwo := 0;
p(q,dummy)
end.
```

6.11 Hardware representation. The representation for tokens and separators given in 6.1 constitutes a reference representation. In addition to these symbols several alternative symbols shall be defined. A processor shall accept all the reference symbols and all the alternative symbols except for any symbol whose representation contains a character not available in the character set of the processor. The reference symbols and the alternative symbols are

given in table 7.

Table 7. Alternative symbols

Reference Symbol	Alternative Symbol
↑	@ or ^
{	(*
}	*)

NOTE. The alternative comment delimiters are equivalent to the reference comment delimiters, thus a comment may begin with "{" and close with "*)", or begin with "(*" and close with "}".

APPENDIX A. COLLECTED SYNTAX

```
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" .

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" .

identifier = letter {(letter | digit)} .

directive = letter {(letter | digit)} .

digit-sequence = digit {digit} .
unsigned-integer = digit-sequence .
unsigned-real =
    unsigned-integer "." digit-sequence ["e" scale-factor] |
    unsigned-integer "e" scale-factor .
unsigned-number = unsigned-integer | unsigned-real .
scale-factor = signed-integer .
sign = "+" | "-" .
signed-integer = [sign] unsigned-integer .
signed-real = [sign] unsigned-real .
signed-number = signed-integer | signed-real .

label = digit-sequence .

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

```
string-element = apostrophe-image | string-character .
apostrophe-image = "''" .
string-character =
    one-of-an-implementation-defined-set-of-characters .


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) ";"} .

statement-part = compound-statement .

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

type-definition = identifier "=" type-denoter .
type-denoter = type-identifier | new-type .
new-type = simple-type | structured-type | pointer-type .

simple-type-identifier = type-identifier .
structured-type-identifier = type-identifier .
pointer-type-identifier = type-identifier .
type-identifier = identifier .

simple-type = ordinal-type | real-type .
ordinal-type = enumerated-type | subrange-type |
                integer-type | Boolean-type | char-type |
                ordinal-type-identifier .

enumerated-type = "(" identifier-list ")" .
identifier-list = identifier { "," identifier } .

subrange-type = constant ".." constant .

structured-type = ["packed"] unpacked-structured-type |
                    structured-type-identifier .
```

```
unpacked-structured-type = array-type ¦ record-type ¦ set-type ¦
                           file-type .

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

record-type = "record"[ field-list [";"]] "end" .
field-list = fixed-part [ ";" variant-part ] ¦ variant-part .
fixed-part = record-section { ";" record-section } .
record-section = identifier-list ":" type-denoter .
variant-part = "case" variant-selector "of"
               variant { ";" variant } .
variant-selector = [tag-field ":"] tag-type .
tag-field = identifier .
variant = case-constant-list ":"  "(" [ field-list [";"] ] ")" .
tag-type = ordinal-type-identifier .
case-constant-list = case-constant { "," case-constant } .
case-constant = constant .

set-type = "set" "of" base-type .
base-type = ordinal-type .

file-type = "file" "of" component-type .

pointer-type = "↑" domain-type ¦ pointer-type-identifier .
domain-type = type-identifier .

variable-declaration = identifier-list ":" type-denoter .

variable-access = entire-variable ¦ component-variable ¦
          referenced-variable ¦ buffer-variable  .

entire-variable = variable-identifier .
variable-identifier = identifier .

component-variable = indexed-variable ¦ field-designator .

indexed-variable =
       array-variable "[" index-expression
       { "," index-expression } "]" .
array-variable = variable-access .

field-designator = record-variable "." field-identifier .
record-variable = variable-access .
field-identifier = identifier .

buffer-variable = file-variable "↑" .
file-variable = variable-access .

referenced-variable = pointer-variable "↑" .
pointer-variable = variable-access .
```

```
procedure-declaration =
         procedure-heading ";" directive |
         procedure-identification ";" procedure-block |
         procedure-heading ";" procedure-block .
procedure-heading =
         "procedure" identifier [ formal-parameter-list ]  .
procedure-identification =
         "procedure" procedure-identifier .
procedure-identifier = identifier .
procedure-block = block .

function-declaration =
        function-heading ";" directive |
        function-identification ";" function-block |
        function-heading ";" function-block .
function-heading =
        "function" identifier [[formal-parameter-list]
        ":" result-type]  .
function-identification =
        "function" function-identifier .
function-identifier = identifier .
result-type = simple-type-identifier |
              pointer-type-identifier .
function-block = block .

formal-parameter-list =
       "(" formal-parameter-section
       {";" formal-parameter-section} ")" .
formal-parameter-section =
       value-parameter-specification |
       variable-parameter-specification |
       procedural-parameter-specification |
       functional-parameter-specification .
value-parameter-specification =
       identifier-list ":" type-identifier .
variable-parameter-specification =
       "var" identifier-list ":"
       (type-identifier | conformant-array-schema) .
conformant-array-schema =
       "array" "[" index-type-specification
       { ";" index-type-specification } "]" "of"
       ( type-identifier | conformant-array-schema ) .
index-type-specification =
       identifier ".." identifier
       ":" ordinal-type-identifier .
bound-identifier = identifier .
procedural-parameter-specification =
       procedure-heading .
functional-parameter-specification =
       function-heading .

unsigned-constant = unsigned-number | character-string |
                    constant-identifier | "nil" .
```

```
factor = variable-access | unsigned-constant | bound-identifier |
         function-designator | set-constructor |
         "(" expression ")" | "not" factor .
set-constructor = "[" [ member-designator
         { "," member-designator } ] "]" .
member-designator = expression [ ".." expression ] .
term = factor { multiplying-operator factor } .
simple-expression = [ sign ] term { adding-operator term } .
expression =
    simple-expression [ relational-operator simple-expression ] .


multiplying-operator = "*" | "/" | "div" | "mod" | "and" .

adding-operator = "+" | "-" | "or" .

relational-operator =
       "=" | "<>" | "<" | ">" | "<=" | ">=" | "in" .

function-designator = function-identifier
                          [ actual-parameter-list ] .
actual-parameter-list =
       "(" actual-parameter { "," actual-parameter } ")" .
actual-parameter =  expression | variable-access |
                      procedure-identifier |
                      function-identifier .

statement = [ label ":" ] ( simple-statement |
                            structured-statement ) .

simple-statement =
       empty-statement | assignment-statement |
       procedure-statement | goto-statement .
empty-statement = .

assignment-statement =
    ( variable-access | function-identifier ) ":=" expression .

procedure-statement = procedure-identifier
                        [ actual-parameter-list ] .

goto-statement = "goto" label .

structured-statement =
       compound-statement | conditional-statement|
       repetitive-statement | with-statement .

compound-statement = "begin" statement-sequence "end" .
statement-sequence = statement { ";" statement } .


conditional-statement = if-statement | case-statement .
```

```
if-statement = "if" Boolean-expression "then" statement
               [ else-part ] .
else-part = "else" statement .

case-statement =
       "case" case-index "of"
       case-list-element {";" case-list-element } [";"] "end" .
case-list-element = case-constant-list ":" statement .
case-index = expression .

repetitive-statement = repeat-statement |
                       while-statement | for-statement .

repeat-statement = "repeat" statement-sequence
                   "until" Boolean-expression .

while-statement = "while" Boolean-expression "do" statement .

for-statement = "for" control-variable ":=" initial-value
                ( "to" | "downto" ) final-value "do" statement .
control-variable = entire-variable .
initial-value = expression .
final-value = expression .

with-statement =
       "with" record-variable-list "do"
       statement .
record-variable-list =
       record-variable { "," record-variable } .

read-parameter-list =
    "("[file-variable ","] variable-access
    {"," variable-access}")" .

readln-parameter-list =
    ["(" (file-variable | variable-access)
    {"," variable-access} ")"] .

write-parameter-list =
    "("[file-variable ","] write-parameter
    {"," write-parameter}")" .
write-parameter =
    expression [":" expression [":" expression ] ] .

writeln-parameter-list =
    ["(" (file-variable | write-parameter)
    {"," write-parameter}")"] .

program = program-heading ";" program-block "." .
program-heading =
       "program" identifier [ "(" program-parameters ")" ] .
program-parameters = identifier-list .
program-block = block .
```

APPENDIX B. INDEX

Pascal Standardisation
A M Addyman
Department of Computer Science
University of Manchester
Oxford Road
MANCHESTER M13 9PL
United Kingdon

HISTORY

In 1977 a working group was formed within the British Standards Institution (BSI) to produce a standard for the programming language Pascal. This working group is responsible to the technical committee on programming languages (DPS/13) and is designated DPS/13/4.

The Attention List

The working group first produced a list of all the known problems with the current definition of Pascal. This was called the Attention List. The final version of the Attention List, which was produced in January 1978, ran to 17 pages. It contained contributions from members of the group, from correspondents and from published criticisms of Pascal.

In April 1978 it was decided that further work on the Attention List should be suspended in favour of the production of a draft. To date there have been five working drafts.

The First Working Draft

The first working draft was produced by dividing up the subject matter into some 16 sections. Many of the section topics corresponded to those of the Revised Report. Each section was the responsibility of two group members; one to write the section and one to comment upon it. The first draft was completed by mid-July 1978.

The Second Working Draft

Immediately following receipt of all the sections which formed the first working draft, the convenor (A.M.Addyman) was a participant at a workshop which was organised to discuss Pascal. The first working draft was the subject of informal discussions at the workshop. On returning from the workshop, the document was redrafted to:

(a) remove obvious inconsistencies in style etc.
(b) produce a document whose form was closer to that of a British Standard.
(c) incorporate comments from the informal discussions.

Both the first and second working drafts were distributed to the members of the working group during August 1978.

The Third Working Draft

A meeting of the group was held in September 1978 to discuss the drafts. This resulted in a list of corrections and amendments to the second working draft. These were incorporated in a new draft together with some further editorial changes which were necessary to form a British Standard. This draft was published in Pascal News #14 and Software Practice and Experience, Volume 9 Number 5 (May 1979).

The Draft for Public Comment

The draft for public comment was an edited version of the third working draft. The changes introduced bring the document into line with BSI/ISO editorial practice. The draft for public comment became BSI document 79/60528DC and ISO/TC97/SC5 N462. It also appeared in IEEE Computer Volume 12 Number 4 (April 1979)

The Fourth Working Draft

The fourth working draft was produced in repsonse to the comments received by August 1979. The group (DPS/13/4) met in September 1979 to discuss these comments and create the draft. This document was circulated as ISO/TC97/SC6 N510. It was further discussed at a November meeting.

The Fifth Working Draft

This draft was the result of an ISO Pascal Experts Group meeting in Turin, Italy in November1979. It was circulated to members of the group in December 1979 for editorial and typographical comments. It became ANSI X3J9/80-003.

The First ISO Draft Proposal

This is the fifth working draft with editorial corrections etc. It is being circulated to those ISO members bodies which are members for ISO/TC97/SC5 (Programming Languages) for comment and three month letter ballot. To ensure the widest possible exposure of the draft it is being printed in SIGPLAN Notices.

Commenting on the Draft Proposal

Ideally you should send your comments to the standards organisation which participates in ISO activities on your country's behalf. All these organisations will have received a copy of the draft and should be actively considering it. In several countries e.g. France, Germany, the Netherlands, the U.K. and the U.S.A. a group was formed which met and considered a previous public draft. In some other countries e.g. Canada, there is the possibility that such a group will soon be formed. If your national standards organisation needs comments from individuals to assist it with its national response you should send your comments to them. In the U.S.A. please send your comments to the secretary of the Joint ANSI X3J9 - IEEE Pascal Standards Committee, who is:

Carol Sledge
On-Line Systems, Inc.
115 Evergreen Height Drive
Pittsburgh PA 15229

Note

The last meeting of the Joint Pascal Committee during the comment/ ballot period will be April 23-25th in Washington D.C.

If you are in the U K, or are outside the USA and are unwilling or unable to communicate with your own standards working body you should send your comments to:

Tony Addyman
(Convener ISO/TC97/SC5/WG4)
Department of Computer Science
University of Manchester
Oxford Road
MANCHESTER M13 9PL
United Kingdom.

Organising your Comments

The previous comment period produced a very large volume of comments in a wide variety of formats. It will assist in the processing of the comments if the following guidelines are adhered to.

1. References to the text should be by section number not by page number.
2. A clear statement of the nature of the problem should be given.
3. Suggestions entailing revision to the text should indicate the preferred wording.

For Example (from the US comments on N462)

Section 6.8.2.2 Assignment-statements

Problem

If the selection of a variable involves the selection of a field of a variant record, for consistency the time of selection should also be implementation-dependent.

Solution

Change line 8 to

"If the selection of the variable involves the indexing of an array, the selection of a field of a variant record,"

Thank You.

Note

The ISO/TC97/SC5 members countries are:

P-Members
Canada, China, Finald, France, Germany, Hungary, Italy, Japan, Netherlands, Romania, Spain, Sweden, Switzerland, UK, USA.

O-Members
Australia, Austria, Belgium, Bulgaria, Czechoslovakia, Denmark, India, Israel, Norway, Poland, Portugal, Republic of South Africa, USSR, Yugoslavia.

5th February 1980