PASCAL USER'S GROUP

# Pascal News

NUMBER 14

COMMUNICATIONS ABOUT THE PROGRAMMING LANGUAGE PASCAL BY PASCALERS

JANUARY, 1979

## TABLE OF CONTENTS

# POLICY: PASCAL NEWS (78/10/01)

* 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! and (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 than we can do."

* An attempt is made to produce Pascal News 3 or 4 times during an academic year from July 1 to June 30; usually September, November, February, and May.

* ALL THE NEWS THAT FITS, 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 - tries to explain 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.

* Volunteer editors are (addresses in the respective sections of Pascal News):

  Andy Mickel - editor
  Jim Miner, Tim Bonham, and Scott Jameson - Implementation Notes editors
  Sara Graffunder and Tim Hoffmann - Here and There editors
  Rich Stevens - Books and Articles editor
  Rich Cichelli - Applications editor
  Tony Addyman and Rick Shaw - Standards editors
  Scott Bertilson, John Easton, Steve Reisman, and Kay Holleman - Tasks editors

PASCAL USER'S GROUP

USER'S

GROUP

# ALL-PURPOSE COUPON

**********

(78/10/01) • •

Pascal User's Group, c/o Andy Mickel
University Computer Center: 227 EX
208 SE Union Street
University of Minnesota
Minneapolis, MN 55455 USA

← *Clip, photocopy, or*
←
← *reproduce, etc. and*
←
← *mail to this address.*

/ / Please enter me as a new member of the *PASCAL USER'S GROUP* for ___ Academic year(s) ending June 30, _____ (not past 1982). I shall receive all the issues of *Pascal News* for each year. Enclosed please find _____. (* Please see the POLICY section on the reverse side for prices and if you are joining from overseas, check for a PUG "regional representative." *)

/ / Please renew my membership in *PASCAL USER'S GROUP* for _____ Academic year(s) ending June 30, _____ (not past 1982). Enclosed please find _____.

/ / Please send a copy of *Pascal News* Number(s) _____. (* See the *Pascal News* POLICY section on the reverse side for prices and issues available. *)

/ / My new address / phone is printed below. Please use it from now on. I'll enclose an old mailing label if I can find one.

/ / You messed up my address / phone. See below.

(* The U.S. Postal Service does not forward *Pascal News*. *)

/ / Enclosed please find a contribution (such as what we are doing with Pascal at our computer installation), idea, article, or opinion which I wish to submit for publication in the next issue of *Pascal News*. (* Please send bug reports to the maintainer of the appropriate implementation listed in the *Pascal News* IMPLEMENTATION NOTES section. *)

/ / None of the above. _____

_____

_____

_____

Other comments:

From: name _____

mailing address _____

_____

_____

_____

phone _____

computer system(s) _____

date _____

(* Your phone number aids communication with other PUG members. *)

POLICY

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 (checks payable to "Pascal User's Group"); we will not bill you.
- please do not send us purchase orders; we cannot endure the paper work! (If you are trying to get your organization to pay for your membership, think of the cost of paperwork involved for such a small sum as a PUG membership!)
- when you join PUG anytime within an academic year: July 1 to June 30, you will receive all issues of Pascal News for that year unless you request otherwise.
- please remember that PUG is run by volunteers who don't consider themselves in the "publishing business." 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, rather than having to answer individually every letter and phone call. We desire to minimize paperwork, because we have other work to do.

- American Region (North and South America): Join through PUG(USA). Send $6.00 per year to the address on the reverse side. International telephone: 1-612-376-7290.

- 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/ Matnematics Department/ The University/ Southampton SO9 5NH/ United Kingdom. International telephone: 44-703-559122 x700.

- Australasian Region (Australia, East Asia - incl. Japan): Join through PUG(AUS). Send $A8.00 per year to: Pascal Users Group/ c/o Arthur Sale/ Dept. of Information Science/ University of Tasmania/ Box 252C GPO/ Hobart, Tasmania 7001/ Australia. International Telephone: 61-02-23 0561.

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? (Costs the same as joining.)

- 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 to help keep us honest. Renewing for more than one year saves us time.

ORDERING BACKISSUES OR EXTRA ISSUES?

- our unusual policy of automatically sending all issues of Pascal News to anyone who joins within an 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, 2, 3, and 4 (January, 1974 - August, 1976) are out of print.
- Issues 5, 6, 7, and 8 (September, 1976 - May, 1977) are out of print. (A few copies of issue 8 remain at PUG(UK) available for £2 each.)
- Issues 9, 10, 11, and 12 (September, 1977 - June, 1978) are available from PUG(USA) all for $10 and from PUG(AUS) all for $A10.

- extra single copies of new issues (current academic year) are: $3 each - PUG(USA); £2 each - PUG(UK); and $A3 each - PUG(AUS).

SENDING MATERIAL FOR PUBLICATION?

- check the addresses for specific editors in Pascal News. Your experiences with Pascal (teaching and otherwise), ideas, letters, opinions, notices, news, articles, conference announcements, reports, implementation information, applications, etc. are welcome. "All The News That Fits, We Print." Please send material single-spaced and in camera-ready (use a dark ribbon and lines 18.5 cm wide) form.
- remember: All letters to us will be printed unless they contain a request to the contrary.

MISCELLANEOUS INQUIRIES?

- Please remember that we will use Pascal News as the medium to answer all inquiries, and we regret to be unable to answer individual requests.

UNIVERSITY OF MINNESOTA
TWIN CITIES

University Computer Center
227 Experimental Engineering Building
Minneapolis, Minnesota 55455
(612) 376-7290

The Draft Pascal Standard

We're devoting a whole issue to this BSI/ISO Working Draft 3 for Standard Pascal. (BSI is the British Standards Institute; ISO is the International Standards Organization.) As Tony Addyman says in his "covering note", the draft is presented for public comment, and comments should be sent to him. When the final draft is submitted to BSI and approved, it will be disseminated through ISO to member bodies such as ANSI (the American National Standards Institute) for adoption. An ISO Standard will avoid the horror of national variants for Pascal.

Pascal standards have been a topic in every issue of Pascal News since issue #6. The PUG membership through Tony will soon benefit from the standardization of Pascal in a form preserving the Revised Report. For an important programming language, this is an unusual event, because it will now spur on manufacturer interest in Pascal.

We have always contended that Pascal standards should be given special consideration, because the language and its development have been unique:

1. Pascal was designed by a single computer scientist--Niklaus Wirth--not by a committee inside or outside a computer manufacturer.
2. Pascal has been used widely and successfully not only to teach the art of programming, but also as an acceptable systems-implementation language.
3. Pascal incorporates machine-independent programming concepts with the goal of program portability. It is an increasingly-used, respectable vehicle for writing portable, systems software. Unlike other programming languages, a clear distinction was made between the language Pascal and any particular implementation of Pascal.
4. Toward this end, there are aspects of Pascal which are explicitly left up to an implementation to define, and there may be cases where an individual implementation may add machine-dependent extensions.
5. Pascal represents a combination of design compromises whose balance was well-considered: simplicity, power, generality, efficiency, portability, clarity, conciseness, redundancy, and robustness. In the late 60's and early 70's, ideas in programming languages and existing machine designs influenced but did not determine the form of Pascal. There exists a delicate equilibrium among these conflicting design goals.

It is important, then, that the BSI/ISO Standard was not meant to incorporate any change to the language with the single exception that the formal parameters of procedures and functions which are themselves parameters be fully specified. The results of the International Working Group on Pascal Extensions (see Pascal News #13) will be included as a non-binding, supplemental Appendix to the Standard.

Finally, it seems only appropriate that a language with European origins has been standardized through the efforts of Europeans: The British Standards Working Group DPS/13/4, The Swedish Technical Committee on Pascal, the French AFCET Subgroup on Pascal, the Pascal Group within the German ACM, and Niklaus Wirth.

# Editor's Contribution

Andy - 78/12/01

PROFESSOR OF COMPUTER SCIENCE
T. KILBURN, C.B.E., M.A., Ph.D.,
D.Sc., F.I.E.E., F.B.C.S., F.R.S.
ICL PROFESSOR OF COMPUTER ENGINEERING
D. B. G. EDWARDS, M.Sc., Ph.D., M.I.E.E.
PROFESSOR OF COMPUTING SCIENCE
F. H. SUMNER, Ph.D., F.B.C.S.
PROFESSOR OF COMPUTER PROGRAMMING
D. MORRIS, Ph.D.


**DEPARTMENT OF COMPUTER SCIENCE**
**THE UNIVERSITY**
**MANCHESTER**
**M13 9PL**

Telephone: 061-273 5466

24th November, 1978.

Dear Andy,

Since sending out a large number of copies of the third working draft, one of the members of DPS/13/4 (Brian Wichmann) has noticed a serious, unintentional error on page 13. I am including a corrected version of this page for you but I cannot afford to send corrections to all the forty or so recipients of the draft. This error will be corrected in the BSI/ISO draft and in any other copies I send out. (I currently have none left!)

Please print this letter or draw the essential contents of the letter to your readers for the benefit of those already in receipt of the draft.

Yours sincerely,

A. M. Addyman.

(* Note: the new page 13 has been included in this issue (page 21). *)

Mr. Andy Mickel,
University Computer Center,
227 Exp. Engr.,
University of Minnesota,
East Bank,
Minneapolis,
MN.55455,
U.S.A.

COVERING NOTE

This document has been sent for processing by B.S.I. and will be the basis of a draft for public comment. The official draft will be available through the usual channels. I will be unable to informally circulate the official draft myself. For this reason, together with the further delay caused by the B.S.I. processing, I am sending you a copy of this working draft.

This document has no official status within B.S.I. Any comments concerning this document should be sent to the address below. If you wish to delay your consideration of this matter until the official draft for public comment becomes available you should acquire the official draft from, and send your comments to, an official standards organisation.

I expect that the official draft will be distributed to ISO member bodies.

A.M.Addyman
8.11.78

Address: A.M.Addyman
Dept. of Computer Science
University of Manchester
Oxford Road,
MANCHESTER M13 9PL.

## A Commentary on Working Draft/3

At the September meeting of DPS/13/4 it was agreed that the second working draft with certain corrections would be sent to B.S.I. for processing to form a draft for public comment. This decision does not indicate that the group are completely satisfied with the document. In fact, there are several areas of detail in which many of the group are unhappy with the draft, but feel that we must stay with the currently accepted definitions.

The rest of this document lists the main areas which caused concern and also draws the attention of the reader to items in Working Draft/3 which are likely to be of interest.

Areas of Concern

6.1.3 and 6.6 Should directives be reserved words?

6.2 Should the ordering of the definition and declaration parts be relaxed? This would permit any number of such parts in any order.

6.4.2.1 and 6.5.2.1 Are the array equivalence rules of any benefit?

6.6.4.1.1 Should the use of rewrite be mandatory? Should its omission have a defined effect?

6.6.4.2.3. and 6.9 Should there be default file parameters? To which procedures and functions do these apply?

6.7.1.1 Should DIV have an implementation-dependent effect for negative operands?

6.9 What are the correct definitions of the form of Pascal output? In particular - leading spaces on numbers and strings in a small field width.

6.9.5. Should the page procedure be removed from the definition of Pascal.

Areas to Note

6.3.2.1 The definitions and subsequent uses of the terms error, implementation defined, implementation dependent and undefined.

6.4.2.2 and 6.8.2.4 Concerning the scope rules.

6.4.2.4 Defines the structure of a textfile

6.4.4 and 6.4.5 Define type compatability etc.

6.6.3.2. VAR parameters are defined as having the effect of a reference implementation.

6.6.3.3 and 6.6.3.4 The only language change - the specification of procedural and functional parameters. This change was introduced after repeated requests to do so from Prof. N. Wirth.

6.7 Defines the type of an expression.

6.8.2.3.3 Note the definition of the for-statement.

Table of Contents
---

0. FOREWORD

This standard is designed to promote the portability of Pascal programs among a variety of data processing systems.
0.1 History. The language Pascal was designed by Professor Niklaus Wirth to satisfy two principal aims.

1. To make available a language suitable for teaching programming as a systematic discipline.

2. To define a language whose implementations may be both reliable and efficient on currently available computers.

1. SCOPE

This standard is designed to promote the portability of Pascal programs among a variety of data processing systems. Programs conforming to this standard, as opposed to extensions or enhancements of this standard are said to be written in "Standard Pascal".

This standard establishes

1. The syntax of Standard Pascal.

2. The semantic rules for interpreting the meaning of a program written in Standard Pascal.

3. The form of writing input data to be processed by a program written in Standard Pascal.

4. The form of output data resulting from the use of a program written in Standard Pascal.

This standard does not prescribe

1. 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.

2. The minimal requirements of a data processing system which is capable of supporting an implemention of a processor for Standard Pascal.

3. The set of commands used to control the environment in which a Standard Pascal program exists.

4. The mechanism by which programs written in Standard Pascal are transformed for use by a data processing system.

2. REFERENCES

ISO 2382 : Glossary of terms used in data processing
BS 3527

## 3. DEFINITIONS

For the purposes of this standard the definitions of BS3527 apply together with the following.

error. A violation by a program of the specification of Standard Pascal whose detection normally requires execution of the program.

implementation defined. Those parts of the language which may differ between processors, but which will be defined for any particular processor.

implementation dependent. Those parts of the language which may differ between processors, for which there need not be a definition for a particular processor.

processor. A compiler, interpreter or other mechanism which accepts a program as input.

scope. The text for which the declaration or definition of an identifier or label is valid.

undefined. The value of a variable when the variable does not necessarily have assigned to it a value of its type.

## 4. THE METALANGUAGE

The metalanguage used to define 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. The following table describes the usages of the various meta-symbols.

| Meta-symbol | Meaning |
|---|---|
| = | is defined to be |
| \| | alternatively |
| . | end of definition |
| [x] | 0 or 1 instance of x |
| {x} | 0 or more repetitions 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 lower case names are hyphenated. The juxtaposition of two meta-symbols in a production implies the concatenation of the text they 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 symbols and separators defined in 6.1.

5. COMPLIANCE

5.1 Processors. A conforming processor shall.

1. Accept all of the features of the language specified in clause 6 with the meanings defined in clause 6.

2. Be accompanied by a document which provides a definition of all implementation defined features.

3. Process each occurrence of an error in one of the following ways.
   a) It is stated in the aforementioned document that the error is not detected.
   b) The processor issued a warning that an occurrence of that error was possible.
   c) The processor detected the error.

4. Be accompanied by a document which separately describes any features accepted by the processor which are not specified in clause 6. Such extensions shall be detailed as being 'extensions to the Standard Pascal specified by BS....: 197-'.

A conforming processor should.

1. Be able to reject any program which uses extensions to the language specified in clause 6.

2. Process programs whose interpretation is affected by implementation dependent features in a manner similar to that prescribed for errors.

A conforming processor may include additional pre-defined procedures and/or functions.

5.2 Programs. A conforming program shall.

1. Use only those features of the language specified in clause 6.

2. Not use any implementation dependent feature.

A conforming program should not have its meaning altered by the truncation of its identifiers to eight characters or the truncation of its labels to four digits.

6. THE PROGRAMMING LANGUAGE PASCAL

6.1 Lexical tokens. The lexical tokens which are used to construct Pascal programs are classified into special symbols, identifiers, numbers, labels and character strings. The syntax given in this section describes the formation of these tokens from characters and their separation, and therefore does not adhere to the same rules as the syntax in the rest of this standard.

```
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"|
         "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" .
```

6.1.1 Special symbols. The special symbols are tokens having a fixed meaning; they are used to specify the syntactic structures of the language.

```
special-symbol = "+"|"-"|"*"|"/"|"="|"<"|">"|"["|"]"|
                 "."|","|":"|";"|"↑"|
                 "<>"|"<="|">="|":="|".."| word-symbol .

word-symbol =      "AND"|"ARRAY"|"BEGIN"|"CASE"|"CONST"|"DIV"|
                   "DOWNTO"|"DO"|"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" .
```

Matching upper and lower case letters are equivalent in word-symbols.

6.1.2 Identifiers. Identifiers serve to denote constants, types,

variables, procedures, functions and programs, and fields and tagfields in records. Identifiers are permitted to be of any length. Matching upper and lower case letters are equivalent in identifiers.

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

Examples:

```
X        Rome      gcd       SUM
```

6.1.3 Directives. Directives only occur immediately after procedure-headings or function-headings.

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

6.1.4 Numbers. The usual decimal notation is used for numbers, which are the constants of the data types integer and real (see 6.4.1.1). The letter E preceding the scale factor means "times ten to the power of".

```
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-number = [sign] unsigned-number .
```

Examples:

```
1      +100      -0.1      5E-3      87.35E+8
```

6.1.5 Labels. Labels are unsigned integers and are distinguished by their apparent integral values.

```
label = unsigned-integer .
```

If a statement is prefixed by a label, a goto statement is permitted to refer to it.

6.1.6 Character strings. Sequences of characters enclosed by apostrophes are called character-strings. Character-strings consisting of a single character are the constants of the standard type char (see 6.4.1.1). Character-strings consisting of n (>1) enclosed characters are the constants of the type (see 6.4.2.1)

```
PACKED ARRAY [1..n] OF char
```

If the character string is to contain an apostrophe, then this apostrophe is to be written twice. Consequently the third example below is a constant of type char.

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

Examples:

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

6.1.7 Comments, spaces, and ends of lines. The construct

"{" any-sequence-of-symbols-not-containg-right-brace "}"

is called a comment. The substitution of a space for a comment does not alter the meaning of a program.

Comments, spaces, and ends of lines are considered to be token separators. An arbitrary number of separators are permitted between any two consecutive tokens, or before the first token of a program text. At least one separator is required between any consecutive pair of tokens made up of identifiers, word-symbols, or numbers. Apart from the use of the space character in character strings, no separators occur within tokens.

6.2 Blocks, Locality and Scope. A block consists of the definitions, declarations and statement-part which together form a part of a procedure-declaration, a function-declaration or a program.
All identifiers and labels with a defining occurence in a particular block are local to that block.

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

The label-declaration-part specifies all labels which mark a statement in the corresponding statement-part. Each label marks one and only one statement in the statement-part. The appearance of a label in a label-declaration is a defining occurrence for the block in which the declaration occurs.

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

The constant-definition-part contains all constant-definitions local to the block.

```
constant-definition-part = "CONST" constant-definition ";"
                              {constant-definition ";"} .
```

The type-definition-part contains all type-definitions which are local to the block.

```
type-defintion-part = "TYPE" type-definition ";"
                        {type-definition ";"} .
```

The variable-declaration-part contains all variable-declarations local to the block.

```
variable-declaration-part = "VAR" variable-declaration ";"
                            {variable-declaration ";"} .
```

The procedure-and-function-declaration-part contains all procedure and function declarations local to the block.

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

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

```
statement-part = compound-statement .
```

Local variables have values which are undefined at the beginning of the statement-part.

6.2.1 Scope

(1) Each identifier or label within the block of a Pascal program has a defining occurrence whose scope encloses all corresponding occurrences of the identifier or label in the program text.

(2) This scope is the range for which the occurrence is a defining one, and all ranges enclosed by that range subject to rules(3) and (4) below.

(3) When an identifier or label which has a defining occurrence for range A has a further defining occurrence for some range B enclosed by A, then range B and all ranges enclosed by B are excluded from the scope of the defining occurrence for range A.

(4) An identifier which is a field-identifier may be used as a field-identifier within a field-designator in any range in which a variable of the corresponding record-type is accessible.

(5) The defining occurrence of an identifier or label precedes all corresponding occurrences of that identifier or label in the program text with one exception, namely that a type-identifier T, which specifies the domain of a pointer-type ↑T, is permitted to have its defining occurrence anywhere in the type-definition-part in which ↑T occurs.

(6) An identifier or label has at most one defining occurrence for a particular range.

6.3 Constant definitions. A constant-definition introduces 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 on the left hand side of a constant-definition is its defining occurrence as a constant-identifier for the block in which the constant-definition occurs. The scope of a constant-identifier does not include its own definition.

A constant-identifier following a sign must denote a value of type integer or real.

6.4 Type definitions. A type determines the set of values which variables of that type assume and the operations performed upon them. A type-definition associates an identifier with the type.

```
type-definition = identifier "=" type .
type = simple-type | structured-type | pointer-type .
```

The occurrence of an identifier on the left hand side of a type-definition is its defining occurrence as a type-identifier for the block in which the type-definition occurs. The scope of a type-identifier does not include its own definition, except for pointer-types see 6.1.4.

A type-identifier is considered to be a simple-type-identifier, a structured-type-identifier, or a pointer-type-identifier, according to the type which it denotes.

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

6.4.1 Simple types. All the simple types define ordered sets of values.

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

An ordinal-type-identifier is one which has been defined to denote an ordinal-type. A real-type-identifier is one which has been defined to denote a real-type.

6.4.1.1 Standard simple types. A standard type is denoted by a predefined type-identifier. The values belonging to a standard type are manipulated by means of predefined primitive operations. The following types are standard in Pascal:

integer The values are a subset of the whole numbers, denoted as described in 6.1.4. The predefined integer constant maxint, whose value is implementation defined, defines the subset of the integers available in any implementation

over which the integer operations are defined.
The range is the set of values:
-maxint, -maxint+1, ... -1, 0, 1, ...maxint-1, maxint.

real — The values are an implementation defined subset of the real numbers denoted as defined in 6.1.4.

Boolean — The values are truth values denoted by the identifiers false and true, such that false is less than true.

char — The values are an implementation defined set of characters. The denotation of character values is described in 6.1.6. The ordering properties of the character values are defined by the ordering of the (implementation defined) ordinal values of the characters, i.e. the relationship between the character variables c1 and c2 is the same as the relationship between ord(c1) and ord(c2). In all Pascal implementations the following relations hold:

(1) The subset of character values representing the digits 0 to 9 is ordered and contiguous.

(2) The subset of character values representing the upper-case letters A to Z is ordered but not necessarily contiguous.

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

Integer, Boolean and char are ordinal-types. Real is a real-type.

Operators applicable to standard types are defined in 6.7.

6.4.1.2 Enumerated types. An enumerated-type defines an ordered set of values by enumeration of the identifiers which denote these values. The ordering of these values is determined by the sequence in which the constants are listed.

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

The occurrence of an identifier within the identifier-list of an enumerated-type is its defining occurrence as a constant for the block in which the enumerated-type occurs.

Examples:

```
(red,yellow,green,blue)
(club,diamond,heart,spade)
(married,divorced,widowed,single)
```

6.4.1.3 Subrange types. The definition of a type as a subrange of

another ordinal-type, called the host type, necessitates identification of the least and the largest value in the subrange. The first constant specifies the lower bound,which is less than or equal to the upper bound.

```
subrange-type = constant ".." constant .
```

Examples:

```
1..100
-10..+10
red..green
```

A variable of subrange-type possesses all the properties of variables of the host type, with the restriction that its value is in the specified closed interval.

6.4.2 Structured types. A structured-type is characterised by the type(s) of its components and by its structuring method. If the component type is itself structured, the resulting structured-type exhibits several levels of structuring.

```
structured-type = ["PACKED"] unpacked-structured-type |
                  structured-type-identifier .
unpacked-structured-type = arraytype | set-type | file-type |
                           record-type .
```

The use of the prefix PACKED in the definition of a structured-type indicates to the processor that storage should be economised, even if this causes an access to a component of a variable of the type to be less efficient in terms of space or time.
An occurrence of the PACKED prefix only affects the representation of the level of the structured-type whose definition it precedes. If a component is itself structured the component's representation is packed only if the PACKED prefix occurs in the definition of its type as well.

6.4.2.1 Array types. An array-type is a structured-type consisting of a fixed number of components which are all of one type, called the component-type. The elements of the array are designated by indices, which are values of the index-type. The array type definition specifies both the index-type and the component-type.

```
array-type = "ARRAY" "[" index-type { "," index-type } "]" "OF"
             component-type .
index-type = ordinal-type .
component-type = type .
```

Examples:

```
ARRAY[1..100] OF real
ARRAY[Boolean] OF colour
```

If the component-type of an array-type is also an array-type, an abbreviated form of definition is permitted. The abbreviated form is equivalent to the full form.

For example:

```
  ARRAY[Boolean] OF
        ARRAY[1..10] OF ARRAY[size] OF real
```

is equivalent to

```
  ARRAY[Boolean,1..10,size] OF real
```

and

```
  PACKED ARRAY[1..10] OF
        PACKED ARRAY[1..8] OF Boolean
```

is equivalent to

```
  PACKED ARRAY[1..10,1..8] OF Boolean
```

The term string type is a generic term used to describe any type which is defined to be

```
  PACKED ARRAY[1..n] OF char
```

6.4.2.2 Record types. A record-type is a structured-type consisting of a fixed number of components, possibly of different types. The record-type definition specifies for each component, called a field, its type and an identifier which denotes it. The occurrence of an identifier as a tag-field or within the identifier-list of a record-section is its defining occurrence as a field-identifier for the record-type in which the tag-field or record-section occurs.

The syntax of a record-type permits the specification of a variant-part. This enables different variables, although of identical type, to exhibit structures which differ in the number and/or types of their components. The variant-part provides for the specification of an optional tag-field. The value of the tag-field indicates which variant is assumed by the record-variable at a given time. Each variant is introduced by one or more constants. All the case-constants are distinct and are of an ordinal-type which is compatible with the tag-type.(see 6.4.4)

For a record with a tag-field a change of variant occurs only when a value associated with a different variant is assigned to the tag-field. At that moment fields associated with the previous variant cease to exist, and those associated with the new variant come into existence, with undefined values. An error is caused if a reference is made to a field of a variant other than the current variant.

For a variant record without a tag-field a change of variant is implied by reference to a field which is associated with a new variant. Again fields associated with the previous variant cease to exist and those associated with the new variant come into existence with undefined values.

```
  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 .
  variant-part = "CASE" [tag-field ":" ] tag-type "OF"
                  variant { ";" variant } .
  tag-field = identifier .
  variant = case-constant-list ":"  "(" [ field-list [";"] ] ")" .
```

```
  tag-type = ordinal-type-identifier .
  case-constant-list = case-constant { "," case-constant } .
  case-constant = constant .
  field-identifier = identifier .
```

Examples:

```
RECORD
  year : integer;
  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 s : shape OF
  triangle : ( side : real;
                  inclination, angle1, angle2 : angle);
  rectangle : (side1, side2 : real;
                  skew, angle3 : angle);
  circle : (diameter : real);
END
```

6.4.2.3 Set types. A set-type defines the range of values which is the powerset of its base type. The largest and smallest values permitted in the base type of a set-type are implementation defined.

```
  set-type = "SET" "OF" base-type .
  base-type = ordinal-type .
```

Operators applicable to set-types are defined in section 6.7.1.3.

6.4.2.4 File types. A file-type is a structured-type consisting of a sequence of components which are all of one type. The number of components, called the length of the file, is not fixed by the file-type definition. A file with zero components is empty.

```
  file-type = "FILE" "OF" type .
```

A standard file-type is provided, which is denoted by the predefined type-identifier text. Variables of type text are called textfiles. Each component of a textfile is of type char, but the sequence of characters represented as a textfile is substructured into lines. All operations applicable to a variable of type FILE OF char are applicable to textfiles, but certain additional operations are also applicable, as described in 6.9. Using the notation of clause 4, the

structure of textfiles is defined as:

text-structure = { {character} linemarker } .

6.4.3 Pointer types. A pointer type consists of an unbounded set of values pointing to variables of a type. No operators are defined on pointers except the tests for equality and inequality.

Pointer values are created by the standard procedure new (see 6.6.4.1.2). The pointer value NIL belongs to every pointer type; it does not point to a variable.

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

6.4.4 Identical and compatible types. Types which are designated at two or more different places in the program text are identical if the same type identifier is used at these places, or if different identifiers are used which have been defined to be equivalent to each other by type definitions of the form T1 = T2.

Two types are compatible if they are identical, or if one is a subrange of the other, or if both are subranges of the same type, or if they are string types with the same number of components, or if they are set types of compatible base types.

6.4.5 Assignment-compatibility. An expression E of type T2 is assignment-compatible with a type T1 if any of the five statements which follow is true.

1. T1 and T2 are identical and neither is a file-type nor a structured-type with a file component.

2. T1 is a real-type and T2 is integer.

3. T1 and T2 are compatible ordinal-types and the value of E is in the closed interval specified by the type T1.

4. T1 and T2 are compatible set-types and all the members of the value of the set E are in the closed interval specified by the base type of T1.

5. T1 and T2 are compatible string types.

At any place where the rule of assignment-compatibility is used.
If T1 and T2 are compatible ordinal-types and the value of the expression E is not in the closed interval specified by the type T1, an error occurs.
If T1 and T2 are compatible set-types and any of the members of the set expression E is not in the closed interval specified by the base type of the type T1, an error occurs.

6.4.6 Example of a type definition part

```
TYPE
  count = integer;
  range = integer;
  colour = (red, yellow, green, blue);
  sex = (male, female);
```

```
year = 1900..1999;
shape = (triangle, rectangle, circle);
card = ARRAY[1..80] OF char;
str = FILE OF char;
polar = RECORD r : real; theta : angle END;
person = ↑persondetails;
persondetails = RECORD
          name, firstname : str;
          age : integer;
          married : Boolean;
          father, child, sibling : person;
          CASE s : sex OF
          male : (enlisted,bearded : Boolean);
          female : (pregnant : Boolean)
END;
tape = FILE OF persondetails;
intfile = FILE OF integer;
```

With the above examples count, range and integer denote identical types. The type year is compatible with, but not identical to, the types range, count and integer.

6.5 Declarations and denotations of variables. A variable declaration consists of a list of identifiers denoting the new variables, followed by their type.

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

The occurrence of an identifier within the identifier-list of a variable-declaration is its defining occurrence as a variable-identifier for the block in which the declaration occurs. A variable declared in a variable-declaration exists during the entire execution process of the block in which it is declared.

Examples:

```
x,y,z: 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 tape
```

A denotation of a variable designates either an entire-variable, a component of a variable, or a variable referenced by a pointer (see 6.4.3). Variables occurring in examples in subsequent chapters are assumed to be declared as indicated above.

```
variable = entire-variable | component-variable |
             referenced-variable .
```

6.5.1 Entire variables. An entire-variable is denoted by its identifier.

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

6.5.2 Component variables. A component of a variable is denoted by the variable followed by a selector specifying the component. The form of the selector depends on the structuring type of the variable.

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

6.5.2.1 Indexed variables. A component of a variable of array-type is denoted by the variable followed an index expression.

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

The index expression is assignment-compatible with the index-type specified in the definition of the array-type.

Examples:

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

If the component of an array-variable is also an array-variable an abbreviation is permitted. The abbreviated form is equivalent to the full form .

For example:

```
m[i][j]
```

is equivalent to

```
m[i,j]
```

6.5.2.2 Field designators. A component of a variable of record-type is denoted by the record-variable followed by the field-identifier of the component.

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

Examples:

```
p2↑.pregnant
coord.theta
```

6.5.2.3 File buffers. The existence of a file variable f with components of type T implies the existence of a buffer variable of type T. This buffer variable is denoted by f↑ and serves to append components to the file during generation, and to access the file during inspection(see 6.6.4.1.1).
At any time, only the one component of a file variable determined by the current file position is directly accessible. This component is

called the current file component and is represented by the file's buffer variable.

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

6.5.3 Referenced variables

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

A variable allocated by the standard procedure new(see 6.6.4.1.2) exists until it is deallocated by the standard procedure dispose (see 6.6.4.1.2).
If p is a pointer variable which is bound to a type T, p denotes that variable and its pointer value, whereas p↑ denotes the variable of type T referenced by p.
An error is caused if the pointer value is NIL or undefined at the time it is dereferenced.

Examples:

```
p1↑.father
p1↑.sibling↑.bearded
```

6.6 Procedure and function declarations

6.6.1 Procedure declarations. A procedure-declaration associates an identifier with a part of a program so that it can be activated by a procedure-statement.

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

The procedure-heading specifies the identifier naming the procedure and the formal parameters (if any).
The appearance of an identifier in the procedure-heading of a procedure is its defining occurence as a procedure-identifier for the block in which the procedure-declaration occurs.

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

The algorithmic actions to be executed upon activation of the procedure by a procedure-statement are specified by the statement-part of the procedure block.
The use of the procedure-identifier in a procedure-statement within the procedure-block implies recursive execution of the procedure.
The full set of directives permitted after a procedure-heading is implementation dependent. However, to allow the call of a procedure to precede textually its definition a forward declaration is provided. A forward declaration consists of a procedure-heading followed by the directive forward. In the subsequent procedure-declaration the formal-parameter-list is omitted. The forward declaration and the procedure-declaration are local to the

same block. The forward declaration and subsequent procedure-declaration constitute a defining occurrence at the place of the forward declaration.

Examples of procedure declarations:

```
PROCEDURE readinteger (VAR f: text; VAR x: integer) ;
VAR i,j: integer;
BEGIN WHILE f↑ = ' ' DO get(f); i :=0;
    WHILE f↑ IN ['0'..'9'] DO
            BEGIN j := ord(f↑)- ord('0');
                i := 10*i + j;
                get(f)
            END;
    x := i
END

PROCEDURE bisect(FUNCTION f(x : real) : real;
                 a,b: real; VAR z: real);
VAR m: real;
BEGIN  {assume f(a) < 0 and f(b) > 0  }
    WHILE abs(a-b) > 1E-10*abs(a) DO
    BEGIN m := (a+b)/2.0;
            IF f(m) < 0 THEN a := m ELSE b :=m
    END;
    z := m
END

PROCEDURE append(VAR f : intfile);
{Enables  items  to  be appended to a file regardless of its current
state}
VAR g : intfile;
    PROCEDURE copy(VAR f,g : intfile);
    BEGIN
    reset(f); rewrite(g);
    WHILE NOT eof(f) DO
            BEGIN
            g↑ := f↑;
            put(g); get(f);
            END;
    END; { of copy }
BEGIN
copy(f,g);
copy(g,f);
END { of append }
```

6.6.2 Function Declarations. Function declarations serve to define parts of the program which compute a value of simple-type or a pointer value. Functions are activated by the evaluation of a function-designator (see 6.7.2) which is a constituent of an expression.

```
function-declaration = function-heading ";"
                       (function-block | directive) .
function-block = block .
```

The function-heading specifies the identifier naming the function, the formal parameters (if any), and the type of the function result.
The appearance of an identifier in the function-heading of a function-declaration is its defining occurrence as a function-identifier for the block in which the function-declaration occurs.

```
function-heading = "FUNCTION" identifier [formal-parameter-list]
                          ":" result-type  .
function-identifier = identifier .
result-type = simple-type-identifier |
              pointer-type-identifier .
```

The algorithmic actions to be executed upon activation of the function by a function-designator are specified by the statement-part of the function block.
The function-block contains at least one assignment-statement which assigns a value to the function-identifier. The result of the function will be the last value assigned. If no assignment occurs the value of the function is undefined. The use of the function-identifier in a function-designator within the function-block implies recursive execution of the function.
The full set of directives permitted after a function-heading is implementation dependent. However, to allow the call of a function to precede textually its definition a forward declaration is provided. A forward declaration consists of a function-heading followed by the directive forward. In the subsequent function-declaration the formal-parameter-list and the result-type are omitted. The forward declaration and the function-declaration are local to the same block. The forward declaration and subsequent function-declaration constitute a defining occurrence at the place of the forward declaration.

Examples:

```
FUNCTION Sqrt(x:real): real;
VAR x0,x1: real;
BEGIN  x1 := x; {x>1, Newton's method }
       REPEAT x0 := x1; x1 := (x0+ x/x0)*0.5
       UNTIL abs(x1-x0) < eps*x1 ;
       Sqrt := x0
END

FUNCTION GCD(m,n : integer) : integer; forward;
```

```
FUNCTION max(a: vector; n: integer): real;
VAR x: real; i: integer;
BEGIN x := a[1];
      FOR i := 2 TO n DO
      BEGIN  {x = max(a[1],....a[i-1])}
      IF x < a[i] THEN x := a[i]
      END;
      {x = max(a[1],....a[n])}
      max := x
END


FUNCTION GCD; {which has been forward declared}
BEGIN IF n=0 THEN GCD := m ELSE GCD := GCD(n,m MOD n)
END


FUNCTION Power(x: real;y: integer): real ; { y >= 0}
VAR w,z: real; i: integer;
BEGIN w := x; z := 1; i := y;
      WHILE i > 0 DO
      BEGIN  {z*(w**i) = x ** y }
            IF odd(i) THEN z := z*w;
            i := i div 2;
            w := sqr(w)
      END;
      {z = x**y }
      Power := z
END
```

6.6.3 Parameters. There are four kinds of parameters : value parameters, variable parameters, procedural parameters and functional parameters. A parameter-group without a preceding specifier is a list of value parameters.

```
formal-parameter-list = "(" formal-parameter-section
                        {";" formal-parameter-section} ")" .
formal-parameter-section =
      ["VAR"] parameter-group |
      procedure-heading |
      function-heading .
parameter-group =
      identifier-list ":" type .
parameter-identifier = identifier .
```

The occurrence of an identifier within the identifier-list of a parameter-group is its defining occurrence as a parameter-identifier for the formal-parameter-list in which it occurs and any corresponding procedure-block or function-block.
The occurrence of an identifier within a procedure-heading in a formal-parameter-section is its defining occurrence as a procedural parameter for the formal-parameter-list in which it occurs and any corresponding procedure-block or function-block.
The occurrence of an identifier within a function-heading in a formal-parameter-section is its defining occurrence as a functional

parameter for the formal-parameter-list in which it occurs and any corresponding procedure-block or function-block.
If the formal-parameter-list is within a procedural parameter or a functional parameter, there is no corresponding procedure-block or function-block.

6.6.3.1 Value parameters. The actual-parameter(see 6.7.2 and 6.8.1.2) is an expression. The formal parameter denotes a variable local to the block. The current value of the expression is assigned to the variable upon activation of the block. The actual-parameter is assignment-compatible with the type of the formal parameter.

6.6.3.2 Variable parameters. The actual-parameter(see 6.7.2 and 6.8.1.2) is a variable. The formal parameter denotes this actual variable during the entire activation of the block. Any operation involving the formal parameter is performed immediately on the actual-parameter. The type of the actual parameter is identical to that of the formal parameter. If the selection of this variable involves the indexing of an array, or the dereferencing of a pointer, then these actions are executed before the activation of the block.

The use of components of variables of any packed type as actual variable parameters is prohibited.

6.6.3.3 Procedural parameters. The actual-parameter(see 6.7.2 and 6.8.1.2) is a procedure-identifier. The formal parameter denotes a procedure which represents the actual procedure during the entire activation of the block. If the procedural parameter, upon activation, accesses any entity non-locally, then the entity accessed is one which was accessible to the procedure when its procedure-identifier was passed as a procedural parameter. The actual procedure and the formal procedure have compatible formal-parameter-lists(see 6.6.3.5).

6.6.3.4 Functional parameters. The actual-parameter(see 6.7.2 and 6.8.1.2) is a function-identifier. The formal parameter denotes a function which represents the actual function during the entire activation of the block. If the functional parameter, upon activation, accesses any entity non-locally, then the entity accessed is one which was accessible to the function when its function-identifier was passed as a functional parameter. The actual function and the formal function have compatible formal-parameter-lists(see 6.6.3.5) and identical result-types.

6.6.3.5 Parameter list compatibility. Two formal-parameter-lists are compatible if they contain the same number of parameters and if the parameters in corresponding positions match. Two parameters match if

They are both value parameters of identical type.

or They are both variable parameters of identical type.

or They are both procedural parameters with compatible parameter lists.

or They are both functional parameters with compatible parameter lists and identical result-types.

6.6.4 Standard procedures and functions. Standard procedures and functions are predeclared in every implementation of Pascal. Any implementation is permitted to feature additional predeclared procedures and functions. Since all predeclared entities are declared in a range surrounding the program, no conflict arises from a declaration redefining the same identifier within the program block.
The standard procedures and functions are listed and explained below.
6.6.4.1 Standard procedures. The effect of using standard procedures as procedural parameters is implementation dependent.

6.6.4.1.1 File handling procedures

put(f) If the predicate eof(f) yields true prior to execution of put(f) then the value of the buffer variable f↑ is appended to the file f, eof(f) remains true and the value of f↑ becomes undefined. If eof(f) does not yield true prior to execution an error occurs.

get(f) If the predicate eof(f) yields false prior to the execution of get(f) then the current file position is advanced to the next component, and the value of this component is assigned to the buffer variable f↑. If no next component exists, then eof(f) becomes true, and the value of f↑ becomes undefined. If eof(f) does not yield false prior to execution, an error occurs.

reset(f) resets the current file position to its beginning and assigns to the buffer variable f↑ the value of the first element of f . eof(f) becomes false, if f is not empty; otherwise the value of f↑ is undefined, and eof(f) is true.
This is a necessary initialising operation prior to reading the file f.

rewrite(f) discards the current value of f such that a new file may be generated. eof(f) becomes true.
This is a necessary initialising operation prior to generating the file f.

If an activation of the procedure put(f) is not separated dynamically from a previous activation of get(f) or reset(f) by an activation of rewrite(f) the effect is implementation dependent.

An error is caused if the current file position of a file f is altered while the buffer variable f↑ is either an actual variable parameter or an element of the record-variable-list of a with-statement or both.

The standard procedures read, write, readln, writeln, and page are described in 6.9.

6.6.4.1.2 Dynamic allocation procedures

new(p) allocates a new variable v and assigns a pointer to v to the pointer variable p. if the type of v is a record type with variants, the form

new(p,t1,....tn) allocates a variable of the variant with tag-field values t1,....,tn. The tag-field values are listed contiguously and in the order of their declaration and are not to be changed during execution from the values indicated. Any trailing tag-fields may be omitted. The tag-field values are not given to the tag-fields by this procedure.

dispose(p) indicates that storage occupied by the variable p↑ is no longer needed. All pointer values which referenced this variable become undefined. If the second form of new was used to allocate the variable then it is necessary to use

dispose(p,t1,....,tn) with identical tag-field values to indicate that storage occupied by this variant is no longer needed.

An error is caused if the value of the pointer parameter of dispose is NIL or undefined.

An error is caused if a variable which is currently either an actual variable parameter, or an element of the record-variable-list of a with-statement, or both, is referred to by the pointer parameter of dispose.

An error is caused if a referenced-variable created using the second form of new is used as an operand in an expression, or the variable in an assignment-statement or as an actual-parameter.

6.6.4.1.3 Transfer procedures

Let the variables a and z be declared by

```
a: ARRAY[m..n] OF T
z: PACKED ARRAY [u..v] OF T
```

where ord(n) - ord(m) >= ord(v) - ord(u)
and ord(m) <= ord(i) <= (ord(n) - ord(v) + ord(u))
then the statement pack(a,i,z) means

```
FOR j := u TO v DO z[j] := a[k]
```

and the statement unpack(z,a,i) means

```
FOR j := u TO v DO a[k] := z[j]
```

where j and k denote auxiliary variables not occuring elswhere in the program and k is the result of applying the function succ (ord(j)-ord(u)) times to i.

6.6.4.2 Standard Functions. The effect of using standard functions as actual functional parameters is implementation dependent.

6.6.4.2.1 Arithmetic Functions. For the following arithmetic functions, the type of the expression x is either real or integer. For the functions abs and sqr, the type of the result is the same as the type of the parameter, x. For the remaining arithmetic functions, the type of the result is always real.

abs(x) computes the absolute value of x.

sqr(x) computes the square of x.

sin(x) computes the sine of x, where x is in radians.

cos(x) computes the cosine of x, where x is in radians.

exp(x) computes the value of the base of natural logarithms raised to the power x.

ln(x) computes the natural logarithm of x, if x is greater than zero. If x is not greater than zero an error occurs.

sqrt(x) computes the positive square root of x, if x is not negative. If x is negative an error occurs.

arctan(x) computes the principal value, in radians, of the arctangent of x.

6.6.4.2.2 Transfer functions

trunc(x) From the real parameter x, this function returns an integer result which is the integral part of x. The absolute value of the result is not greater than the absolute value of the parameter.
An error occurs if the result is not a value of the type integer.
For example:
trunc(3.7) yields 3
trunc(-3.7) yields -3

round(x) From the real parameter x, this function returns an integer result which is the value of x rounded to the nearest integer. If x is positive or zero then round(x) is equivalent to trunc(x+0.5), otherwise round(x) is equivalent to trunc(x-0.5).
An error occurs if the result is not a value of the type integer.

For example:
round(3.7) yields 4
round(-3.7) yields -4

6.6.4.2.3 Ordinal functions

ord(x) The parameter x is an expression of ordinal-type. The result is of type integer. If the parameter is of type integer then the value of the parameter is yielded as the result. If the parameter is type char, the result is implementation defined. If the parameter is of any other ordinal-type, the result is the ordinal number determined by mapping the values to the type on to consecutive non-negative integers starting at zero.
ord(false) yields 0
ord(true) yields 1

chr(x) yields the character value whose ordinal number is equal to the value of the integer expression x, if such a character value exists. If such a character value does not exist an error occurs.
For any character value, ch, the following is true:
chr(ord(ch)) = ch

succ(x) The parameter x is an expression of ordinal-type. The result is of a type identical to that of the expression (see 6.7). The function yields a value whose ordinal number is one greater than that of the expression x, if such a value exists. If such a value does not exist, an error occurs.

pred(x) The parameter x is an expression of ordinal-type. The result is of a type identical to that of the expression (see 6.7). The function yields a value whose ordinal number is one less than that of the expression x, if such a value exists. If such a value does not exist, an error occurs.

6.6.4.2.4 Predicates

odd(x) yields true if the integer expression x is odd otherwise it yields false.

eof(f) indicates whether the associated buffer variable f↑ is positioned at the end of the file f. If the actual-parameter-list is omitted the function is applied to the standard file input.

eoln(f) indicates whether the associated buffer variable f↑ is positioned at the end of a line in the textfile f(see 6.9). If the actual-parameter-list is omitted the function is applied to the standard file input.

6.7 Expressions. Expressions consist of operators and operands i.e. variables, constants, and function designators. An error is caused if any variable or function used as an operand in an expression has an undefined value at the time of its use.

The rules of composition specify operator precedences according to four classes of operators. The operator NOT has 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 are executed from left to right.

```
unsigned-constant = unsigned-number | string |
                    constant-identifier | "NIL" .
factor = variable | unsigned-constant |
         function-designator | set | "(" expression ")" |
         "NOT" factor .
set = "[" [ element { "," element } ] "]" .
element = expression [ ".." expression ] .
term = factor { multiplying-operator factor } .
simple-expression = [ sign ] term { adding-operator term } .
expression =
    simple-expression [ relational-operator simple-expression ] .
```

Any operand whose type is S, where S is a subrange of T, is treated as if it were of type T. Similarly, any operand whose type is SET OF S is treated as if it were of type SET OF T. Consequently an expression which consists of a single operand of type S is itself of type T and an expression which consists of a single operand of type SET OF S is itself of type SET OF T.
Expressions which are members of a set are of identical type, which is the base type of the set. [] denotes the empty set which belongs to every set type. The set [x..y] denotes the set of all values of the base type in the closed interval x to y .
If x is greater than y then [x..y] denotes the empty set.
An error is caused if the value of an expression which is the member of a set is outside the implementation defined limits.

Examples:

Factors:

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

Terms:

```
x*y
i/(1-i)
p OR q
(x<=y) AND (y < z)
```

Simple expressions:

```
x+y
-x
hue1 + hue2
i*j + 1
```

Expressions:

```
x = 1.5
p<=q
p = q AND r
(i<j) = (j<k)
c IN hue1
```

6.7.1 Operators

```
multiplying-operator = "*" | "/" | "DIV" | "MOD" | "AND" .

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

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

The order of evaluation of the operands of a binary operator is implementation dependent.

6.7.1.1 Arithmetic operators

Binary

| operator | operation | type of operands | type of result |
|---|---|---|---|
| + | addition | integer or real | integer or real |
| - | subtraction | integer or real | integer or real |
| * | multiplication | integer or real | integer or real |
| / | division | integer or real | real |
| DIV | division with truncation | integer | integer |
| MOD | modulo | integer | integer |

Unary

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

The symbols +, - and * are also used as set operators (see 6.7.1.3).

If both the operands of the addition, subtraction or multiplication operators are of the type integer, then the result is of the type

integer otherwise the result is of the type real. If the operand of the identity or sign-inversion operators is of the type integer then the result is of the type integer otherwise the result is of the type real.

The value of i DIV j is such that i-j < (i DIV j)*j <= i, if i >= 0 and j > 0; an error is caused if j = 0; otherwise the result is implementation defined.

The value of i MOD j is equal to the value of i - (i DIV j)*j.

The predefined constant maxint is of integer type and has an implementation defined value. This value satisfies the following conditions.

1. All integral values in the closed interval from -maxint to +maxint are representable in the type integer.

2. Any unary operation performed on an integer value in the above interval is correctly performed according to the mathematical rules for integer arithmetic.

3. Any binary integer operation on two integer values in the above interval is correctly performed according to the mathematical rules for integer arithmetic, provided that the result is also in that interval.
   If the result is not in that interval an error occurs.

4. Any relational operation on two integer values in the above interval is correctly performed according to the mathematical rules for integer arithmetic.

6.7.1.2 Boolean operators

| operator | operation | type of operands | type of result |
|---|---|---|---|
| OR | logical "or" | Boolean | Boolean |
| AND | logical "and" | Boolean | Boolean |
| NOT | logical negation | Boolean | Boolean |

Whether a Boolean expression is completely or partially evaluated if its value can be determined by partial evaluation is implementation dependent.

6.7.1.3 Set operators

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

6.7.1.4 Relational operators

| operator | type of operands | type of result |
|---|---|---|
| = <> | any set, simple, pointer or string type | Boolean |
| < > | any simple or string type | Boolean |
| <= >= | any set, simple or string type | Boolean |
| IN | left operand:any ordinal type T<br>right operand: SET OF T | Boolean |

Except when applied to sets the operators <> , <= , >= stand for notequal, less than or equal and greater than or equal respectively.
The operands of =, <>, <, >, >=, and <= are either of compatible type or one operand is real and the other is integer.
If u and v are set operands, u <= v denotes the inclusion of u in v and u >= v denotes the inclusion of v in u.
If p and q are Boolean operands, p = q denotes their equivalence and p <= q denotes the implication of q by p, since false < true.
When the relational operators = , <> , < , > , <= , >= are used to compare strings (see 6.4.2.1), they denote lexicographic ordering according to the ordering of the character set(see 6.4.1.1).
The operator IN yields the value true if the value of the operand of ordinal-type is a member of the set, otherwise it yields the value false. In particular, if the value of the operand of ordinal-type is outside the implementation defined limits, the operator IN yields false.

6.7.2 Function designators. A function-designator specifies the activation of the function denoted by the function-identifier. The function-designator contains a (possibly empty) list of actual-parameters which are substituted in place of their corresponding formal parameters defined in the function-declaration. The correspondence is established by the positions of the parameters in the lists of actual and formal parameters respectively. The

number of actual-parameters is equal to the number of formal parameters. The order of evaluation and binding of the actual-parameters is implementation dependent.

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

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

6.8 Statements. Statements denote algorithmic actions, and are said to be executable. They may be prefixed by a label which can be referenced by goto statements.

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

6.8.1 Simple statements. A simple-statement is a statement of which no part constitutes another statement. An empty statement consists of no symbols and denotes no action.

```
simple-statement =
      [ ( assignment-statement ¦
      procedure-statement ¦ goto-statement ) ] .
```

6.8.1.1 Assignment statements. The assignment-statement serves to replace the current value of a variable by a new value specified as an expression.

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

The expression is assignment-compatible with the type of the variable or function.
If the selection of the variable involves the indexing of an array or the dereferencing of a pointer, then whether these actions precede or follow the evaluation of the expression is implementation dependent.

Examples:
```
x := y+z
p := (1<=i) AND (i<100)
i := sqr(k) - (i*j)
hue1 := [blue,succ(c)]
```

6.8.1.2 Procedure statements. A procedure-statement serves to

execute the procedure denoted by the procedure-identifier. The procedure-statement contains a (possibly empty) list of actual-parameters which are substituted in place of their corresponding formal parameters defined in the procedure-declaration (see 6.6.3). The correspondence is established by the positions of the parameters in the lists of actual and formal parameters respectively. The number of actual-parameters is equal to the number of formal parameters.The order of evaluation and binding of the actual-parameters is implementation dependent.

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

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

6.8.1.3 Goto statements. A goto-statement serves to indicate that further processing is to continue at another part of the program text, namely at the place of the label.

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

The following restrictions hold concerning the use of labels:

1. A goto-statement leading to the label which prefixes a statement S causes an error unless the goto statement is activated either by S or by a statement in the statement-sequence(see 6.8.2.1 and 6.8.2.3.2) of which S is an immediate constituent.

2. A goto-statement does not refer to a case-constant.

6.8.2 Structured statements. Structured-statements are constructs composed of other statements which 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 .
```

6.8.2.1 Compound statements. The compound-statement specifies that its component statements are to be executed in the same sequence as they are written. The symbols BEGIN and END act as statement brackets.

```
compound-statement = "BEGIN" statement-sequence "END" .
statement-sequence = statement { ";" statement } .
```

```
Example:    BEGIN  z := x ; x := y; y := z END
```

6.8.2.2 Conditional statements. A conditional-statement selects for execution a single one of its component statements.

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

6.8.2.2.1 If statements

```
if-statement = "IF" Boolean-expression "THEN" statement
               [ else-part ] .
else-part = "ELSE" statement .
```

If the Boolean-expression yields the value true, the if-statement specifies that the statement following the THEN be executed. If the Boolean-expression yields false the action depends on the existence of an else-part; if the else-part is present the statement following the ELSE is executed otherwise an empty statement is executed.

```
Boolean-expression = expression .
```

A Boolean-expression is an expression which produces a result of type Boolean.

The syntactic ambiguity arising from the construct

```
IF e1 THEN IF e2 THEN s1 ELSE s2
```

is resolved by interpreting the construct as being equivalent to

```
IF e1 THEN
  BEGIN
  IF e2 THEN s1 ELSE s2
  END
```

Examples:

```
IF x < 1.5 THEN z := x+y ELSE z := 1.5
IF p1 <> NIL THEN p1 := p1↑.father
```

6.8.2.2.2 Case statements. The case-statement consists of an expression (the case index) and a list of statements. Each statement is preceded by one or more constants. All the case-constants are distinct and are of an ordinal-type which is identical to that of the case-index. The case-statement specifies that the statement be executed whose case-constant is equal to the current value of the selector.
An error is caused if none of the case-constants is equal to the current value of the selector.

```
case-statement =
       "CASE" expression "OF"
       case-list-element {";" case-list-element } [";"] "END" .
case-list-element = case-constant-list ":" statement .
```

Examples:

```
CASE operator OF          CASE i OF
  plus:   x := x+y;          1: x := sin(x);
  minus:  x := x-y;          2: x := cos(x);
  times:  x := x*y           3: x := exp(x);
END                          4: x := ln(x)
                          END
```

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

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

6.8.2.3.1 Repeat statements

```
repeat-statement = "REPEAT" statement-sequence
                   "UNTIL" Boolean-expression .
```

The sequence of statements between the symbols REPEAT and UNTIL is repeatedly executed until the Boolean-expression yields the value true on completion of the statement-sequence. The statement-sequence is executed at least once, since the Boolean-expression is evaluated after execution of the statement-sequence.

Examples:

```
REPEAT k := i MOD j;
  i := j;
  j := k
UNTIL j = 0

REPEAT
  process(f↑);
  get(f)
UNTIL eof(f)
```

6.8.2.3.2 While statements

```
while-statement = "WHILE" Boolean-expression "DO" statement .
```

The statement is repeatedly executed while the Boolean expression yields the value true. If its value is false at the beginning, the statement is not executed at all.

The while-statement

```
WHILE b DO body
```

is equivalent to

```
IF b THEN
  REPEAT
  body
  UNTIL NOT b
```

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
```

6.8.2.3.3 For statements. The for-statement indicates that a statement is to be repeatedly executed while a progression of values is assigned to a variable which is called 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 is an entire-variable which is local to the immmediately enclosing block. The control-variable is of ordinal-type, and the initial and final value are of a type compatible with this type. An error is caused if the control-variable is assigned to by the repeated statement or altered by any procedure or function activated by the repeated statement. After a for-statement is executed (other than being left by a goto statement leading out of it) the value of the control-variable is left undefined. Apart from the above restrictions

The for-statement

```
FOR v := e1 TO e2 DO body
```

is 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
```

is 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 are auxiliary variables of the host type of the variable v which do not occur elsewhere in the program.

Examples:

```
FOR i := 2 TO 63 DO
  IF a[i] > max THEN max := a[i]

FOR i := 1 TO n DO
  FOR j := 1 TO n DO
    BEGIN
    x := 0;
    FOR k := 1 TO n DO
      x := x + m1[i,k]*m2[k,j];
    m[i,j] := x
    END

FOR c := red TO blue DO q(c)
```

6.8.2.4 With statements

```
with-statement =
       "WITH" record-variable-list "DO"
       statement .
record-variable-list =
       record-variable { "," record-variable } .
variable-identifier = field-identifier .
```

The occurrence of a record-variable in the record-variable-list is a defining occurrence of its field-identifiers as variable-identifiers for the with-statement in which the record-variable-list occurs.

The statement

```
WITH v1,v2, ....vn DO s
```

is 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
```

is equivalent to

```
IF date.month = 12 THEN
  BEGIN date.month := 1; date.year := date.year+1
  END
ELSE date.month := date.month+1
```

If the selection of a variable in the record-variable-list involves the indexing of an array or the dereferencing of a pointer, then these actions are executed before the component statement is executed.

6.9 Input and output. The basis of legible input and output consists of textfiles (see 6.4.2.4) that are passed as program-parameters (see 6.10) to a Pascal program and in its environment represent some input or output device such as a terminal, a card reader, a line printer or a file in the backing store. In order to facilitate the handling of textfiles, the four standard procedures read, write, readln, and writeln are introduced in addition to the procedures get and put. The new procedures are used with a more flexible syntax for their parameter lists, allowing, among other things, for a variable number of parameters. Moreover, the parameters need not necessarily be of type char. Certain other types are permitted, in which case the data transfer is accompanied by an implicit data conversion

operation. If the first parameter is a file variable, then this is the file to be read or written. Otherwise, the standard files input and output are automatically assumed as default values in the cases of reading and writing respectively.(see 6.10)

Textfiles represent a special case among file types insofar as texts are substructured into lines by line markers (see 6.4.2.4). If, upon reading a textfile f, the file position is advanced to a line marker, that is past the last character of a line, then the value of the buffer variable f↑ becomes a space, and the standard function eoln(f) yields the value true. Advancing the file position once more either causes eof(f) to become true or assigns to f↑ the first character of the next line, and eoln(f) yields false (unless the next line consists of zero characters). Line markers, not being elements of type char, can be generated only by the procedure writeln.

6.9.1 The procedure read. The syntax of the parameter list of read is

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

The following rules hold for the procedure read; f denotes a textfile and v1..vn denote variables of the types char (or a subrange of char), integer (or a subrange of integer), or real.

1. read(f,v1,...,vn) is equivalent to

   BEGIN read(f,v1); ... ; read(f,vn) END

2. If v is a variable of type char or subrange of char, then read(f,v) is equivalent to

   BEGIN v := f↑; get(f) END

3. If v is a variable of type integer (or subrange thereof), then read(f,v) implies the reading from f of a sequence of characters which form a signed-integer according to the syntax of 6.1.4. The value of the integer is assigned to v, if it is assignment-compatible with the type of v. Preceding spaces and line markers are skipped. Reading ceases as soon as the file buffer variable f↑ contains a character which does not form part of a signed-integer. An error occurs if the sequence of characters does not form a signed-integer according to the specified syntax.

4. If v is a variable of type real then read(f,v) implies the reading from f of a sequence of characters which form a signed-number according to the syntax of 6.1.4. The value of the number is assigned to the variable v. Preceding spaces and line markers are skipped. Reading ceases as soon as the file buffer

variable f↑ contains a character which does not form part of a signed-number. An error occurs if the sequence of characters does not form a signed-number according to the specified syntax.

The procedure read can also be used to read from a file f which is not a textfile. read(f,x) is in this case equivalent to

```
BEGIN x := f↑; get(f) END
```

6.9.2 The procedure readln. The syntax of the parameter list of readln is

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

1. readln(f,v1,...,vn) is equivalent to

```
BEGIN read(f,v1,...,vn); readln(f) END
```

2. readln(f) is equivalent to

```
BEGIN WHILE NOT eoln(f) DO get(f); get(f) END
```

Readln is used to skip to the beginning of the next line.

6.9.3 The procedure write. The syntax of the parameter list of write is

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

The following rules hold for the procedure write; f denotes a textfile, p1,...,pn denote write-parameters, e denotes an expression, m and n denote expressions of type integer.

1. write(f,p1,...,pn) is equivalent to

```
BEGIN write(f,p1); ... ; write(f,pn) END
```

2. The write-parameters p have the following forms:

```
e:m   e:m:n   e
```

e is an expression whose value is to be "written" on the file f: it may be numeric (integer or real), char, Boolean or a string. m and n are expressions whose integer values are the field-width parameters. Their values are greater than zero. Exactly m characters will be written (with an appropriate number of spaces

to the left of the representation of e), except when e is a numeric value which requires more than m characters for its representation: in such cases the number of characters written will be as small as is consistent with the representation of e (see rules 4 and 5).

write(f,e) is equivalent to write(f,e:m), using a default value for m which depends on the type of e: for integer, Boolean and real types this value is implementation-defined.

write(f,e:m:n) is applicable only if e is of type real (see rule 5).

3. If e is of type char then the default value for m is one.

4. If e is of type integer , then the decimal representation of the number e is written on the file f. If p is the positive integer defined by:

```
IF e = 0
   THEN p := 1
   ELSE determine p such that 10**(p-1) <= abs(e) < 10**p
```

the representation consists of:

(a) if m >= p + 1
(m-p-1) blanks,
the sign character ('-' if e<0, otherwise a blank),
p digits.

(b)If m<p+1, p characters are written if e>=0, (p+1) if e<0.

5. If e is of type real, a decimal representation of the number e, rounded to the specified number of significant figures or decimal places, is written on the file f.

write(f,e:m) causes a floating-point representation of e to be written. If d is an implementation-defined value (representing the number of digit characters written in an exponent), and the non-negative number er and the integer p are defined by:

```
IF e = 0.0
   THEN BEGIN er := 0.0; p := 1 END
   ELSE
   BEGIN
      er := abs(e);
      determine p such that 10**(p-1) <= er < 10**p;
      er := er + 0.5 * (10**(p-m+d+4));
      er is truncated to (m-d-4) significant decimal figures
   END
```

the representation consists of:

(a) if m >= d + 6:
the sign character
('-' if e<0 and er<>0, otherwise a space),
the leading digit of er,
the character '.',
the next (m-d-5) digits of er,
the character 'E',
the sign of (p - 1) ('+' or '-'),
d digits for (p - 1)
(with leading zeros if necessary).

(b)If m<d+6, (d+6) characters are written, including one digit after the decimal point.

write(f, e:m:n) causes a fixed-point representation of e to be written. If the non-negative number er and the positive integer p are defined by:

```
IF e = 0.0
   THEN er := 0.0
   ELSE
   BEGIN
      er := abs(e);
      er := er + 0.5 * 10**(-n);
      er is truncated to n decimal places
   END;
IF trunc(er) = 0
   THEN p := 1
   ELSE determine p such that 10**(p-1) <= trunc(er) < 10**p
```

the representation consists of:

(a) if m >= p+n+2:
(m-p-n-2) spaces,
the sign character
('-' if e<0 and er<>0, otherwise a space),
the first p digits of er,
the character '.',
the next n digits of er.

(b)If m<p+n+2, (p+n+2) characters are written.

6. If e is of type Boolean, then a representation of the word true or the word false (as appropriate) is written on the file f. This is equivalent to

```
write(f,'TRUE ':m)  or  write(f,'FALSE':m)
```

as appropriate (see rule 7.).

7. If e is of a string type, then the value of e is written on the file f preceded by (m-n) spaces if m>=n. If m<n, characters one

through m of the string are written. If m is omitted, the default value is n.

The procedure write can also be used to write on to a file f which is not a textfile. write(f,x) is in this case equivalent to

```
BEGIN f↑ := x;  put(f) END
```

6.9.4 The procedure writeln. The syntax of the parameter list of writeln is

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

1. writeln(f,p1,...,pn) is equivalent to

```
BEGIN write(f,p1,...,pn);  writeln(f) END
```

2. writeln(f) appends a line marker (see 8.2.4) to the file f.

6.9.5 The procedure page

page(f) causes skipping to the top of a new page, when the textfile f is printed. If the actual-parameter-list is omitted the procedure is applied to the standard file output.

6.10 Programs. A Pascal program has the form of a procedure declaration except for its heading.

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

program-heading =
       "PROGRAM" identifier [ "(" program-parameters ")" ] .

program-parameters = identifier-list .
```

The identifier following the symbol PROGRAM is the program name; it has no significance within the program. The program-parameters denote entities that exist outside the program, and through which communication with the environment of the program is possible. These entities (usually files) are called external, and are declared in the variable-declaration-part of the block which constitutes the program. The two standard files input and output are not declared explicitly (see 6.9), but are listed as parameters in the program-heading if they are used in the program. The appearance of the files input or output as program-parameters causes them to be declared in the program block. The initialising statements reset(input) and rewrite(output) are automatically generated if required. The effect of an explicit use of reset or rewrite on the standard files input or output is implementation dependent.

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 line-markers of the
 textfile input to the textfile output.}
VAR ch: char;
BEGIN
   WHILE NOT eof(input) DO
   BEGIN
      WHILE NOT eoln(input) DO
         BEGIN read(ch); write(ch)
         END;
      readln; writeln
   END
END.
```

6.11 Hardware representation. The representation for tokens and separators given in 6.1 constitutes a reference representation.
This is the standard representation and is used for publication or interchange of programs written in Standard Pascal.
Only in cases where the character set is insufficiently rich to permit the full use of the reference language, are the character combinations given in the table below defined as substitute symbols.
In the case of symbols which are pairwise matching, such as comment markers, the pairwise sets are either both be in the reference representation, or both be in an equivalent representation.

| Reference Symbol | Equivalent Symbol |
|---|---|
| ↑ | @ or ^ |
| { } | (* *) |
| [ | (. |
| ] | .) |
| := | .= or %= |
| ; | ., |
| : | % |
| > | GT |
| < | LT |
| >= | GE |
| <= | LE |
| <> | NE |

## PASCAL STANDARDISATION

This is a brief summary of what has been done so far in the effort to produce a standard. Although several individuals have made significant technical contributions, these have been omitted from this summary. This summary is essentially a catalogue of significant events.

| | |
|---|---|
| February 1975 | A. M. Addyman joins DPS/13 (then called DPE/13). A Pascaler is now in a position where action is possible. |
| June 1976 | Professor N. Wirth welcomes the idea of an official standard. |
| August 1976 | In Pascal News (letter) #6 my intentions are made public for support or objections. |
| September 1976 | Pascal is placed on the DPS/13 agenda. |
| March 1977 | Standards debate at the Southampton Symposium (see Pascal News (letter) #8). The symposium acted as a catalyst. |
| April 1977 | Attention List (part 1) created (PN #8). |
| June 1977 | BSI Pascal project is proposed. Formation of DPS/13/4 is approved. |
| September 1977 | BSI Pascal project is approved.<br>Another Attention List is produced (PN #9/10).<br>Initial meeting of DPS/13/4 to discuss the Attention Lists. |
| November 1977 | ISO/TC97/SC5 meeting in The Hague. Pascal is on the agenda at the request of the UK. Sweden indicate that they have some Pascal activity. The UK announces its intention to propose an ISO project. |
| December 1977 | Swedish Attention List received. UK Working papers sent to Sweden. |
| January 1978 | A combined Attention List #2 is produced and circulated. |
| February 1978 | Contact established with Lecarme's French group and with Arthur Sale. |
| March 1978 | After much unnecessary delay a proposal is sent to the SC5 secretariat for voting on the formation of a work item for Pascal. |
| April 1978 | DPS/13/4 decides to prepare its first working draft. |

| | |
|---|---|
| July 1978 | The La Jolla Workshop. This provided an opportunity for Pascalers to discuss the standardisation problems informally. These discussions and a lot of individual effort over August led to the second working draft. |
| September 1978 | DPS/13/4 discusses the working drafts and agrees on a set of amendments. |
| October 1978 | After two voting extensions Pascal is added to the ISO work program. The third working draft is sent to BSI for processing. ANSI announces the formation of X3J9. |

The Future

| | |
|---|---|
| December 1978 | X3J9 to meet in Washington D.C.<br>DPS/13 to meet. BSI amendments to the working draft to be received and processed. |
| December 1978<br>January/February 1979? | The BSI draft for comment will be available from BSI. It will also be sent to the ISO/TC97/SC5 secretariat for distribution and comment. |

## *MEMBERS OF DPS/13/4*

*A. M. Addyman,*
*Department of Computer Science,*
*The University,*
*Oxford Road,*
*Manchester,*
*M13 9PL.*

*061-273 7121 : Ext.5546*

*Convenor, etc.*

*Professor D. W. Barron,*
*Computer Studies Group,*
*The University,*
*Southampton,*
*SO9 5NH.*

*0703-559122 : Ext.700*

*May no longer be active.*

*R. Brewer,*
*Software Department,*
*GEC Computers Limited,*
*Elstree Way,*
*Borehamwood,*
*Herts.*

*D. G. Burnett-Hall,*
*Department of Computer Science,*
*University of York,*
*Heslington,*
*York,*
*YO1 5DD.*

*0904-59861*

R. M. De Morgan,
European Software Engineering,
Digital Equipment Company Limited,
Fountain House,
The Butts Centre,
Reading,
RG1 7QN.

0734-583555

May no longer be active.

W. Findlay,
Computing Science Department,
University of Glasgow,
Glasgow,
G12 8QQ.

041-339 8855 : Ext.7391

M. I. Jackson,
I.D.E.C.,
Holbrook House
Cockfosters Road,
Barnet,
Herts, EN4 0DU.

01-440 4141

(This is ITT Business Systems).

D. A. Joslin,
Hull College of Higher Education,
Inglemure Avenue,
Hull,
HU6 7LJ.

0482-42157

M. J. Rees,
Computer Studies Group,
The University,
Southampton,
SO9 5NH.

0703-559122

May no longer be active.

D. A. Watt,
Computing Science Department,
University of Glasgow,
Glasgow,
G12 8QQ.

041-339 8855 : Ext.7458

J. Welsh
Department of Computer Science,
Queen's University,
Belfast,
BT7 1NN.

0232-45133 : Ext.3221

B. A. Wichmann,
Department of Industry,
National Physical Laboratory,
Teddington,
Middlesex,
TW11 0LW.

01-977 3222

## A Proposal for a New Work Item - Pascal (* Early 1978 *)

This document is to accompany the corresponding ISO PLACO form.

### 1. Introduction

The programming language Pascal is a simple high-level language. It is a general-purpose rather than an all-purpose language. Pascal is being used increasingly in three areas:

1) The writing of system software
2) The writing of application software
3) The teaching of programming.

There are implementations of Pascal available for most mainframe and mini-computers. There are also Pascal implementations for seven different microprocessors.

In the summer of 1976 a Pascal Users Group (PUG) was formed to encourage the use of Pascal. In the first 18 months of its existence, PUG has attracted 1600 members from 30 different countries.

### 2. Description

The purpose of the proposed project is to produce a formal standard for the programming language Pascal. The language is currently, informally, defined by three documents:

| | |
|---|---|
| The Pascal Report | complete references |
| The Pascal User Manual | appear later |
| The Axiomatic Definition of Pascal | |

The definition provided by these documents is incomplete and, in a few instances, contradictory. The project will involve the identification and resolution of these difficulties.

### 3. Benefits

The benefits to be derived from the production of a standard for Pascal are the same as those for any other widely-used programming language, namely:

1) Enhanced program portability reduces programming costs, which are estimated to be about 70% of the cost of a computer system.
2) It will become easier to recruit trained staff and cheaper to train staff.

## 4. Development Feasibility

The most important factor in this proposal is the timeliness of the standardisation of Pascal. Pascal has been implemented on a large number of different computers. In most cases, the implementations have been made compatible with existing compilers for different computers. If the problems relating to the definition of Pascal are not resolved in the very near future, there is a danger that the various implementations will become incompatible. The growth of a large number of incompatibilities would hinder any subsequent standardisation activities.

The current lack of any significant incompatibilities should be seen as a good reason for standardisation now.

### 4.1 Available Resources

There are already two national working groups concerned with the production of a Pascal standard. They are:

| | |
|---|---|
| DPS/13/4 | (in the United Kingdom) |
| Swedish Technical Committee on Pascal | (in Sweden) |

These two groups are cooperating with each other and are corresponding with interested parties in the following countries: Australia, Canada, Denmark, France, Germany, Poland, Switzerland and the USA. Many of these correspondants are suppliers of Pascal compilers.

### 4.2 Cost

This is difficult to estimate. However, since most (if not all) of the deficiencies of the current reference documents are known, there are no technical obstacles to the production of a draft proposal within 12 months.

### 4.3 Bibliographic References

Jensen, K. and Wirth, N. (1975), Pascal - User Manual and Report. (Springer-Verlag, New York)

Hoare, C.A.R. and Wirth, N. (1973), An axiomatic definition of the programming language Pascal, Acta Informatica 2, 335-55

Habermann, A.N. (1974), Critical comments on the programming language Pascal, Acta Informatica 3, 47-57

Lecarme, O. and Desjardins, P. (1975), More comments on the programming language Pascal, Acta Informatica, 4, 231-45

Welsh, J., Sneeringer, W.J., and Hoare, C.A.R. (1977), Ambiguities and insecurities in Pascal, Software Practice and Experience, 7, 685-96

Wirth, N. (1975), An assessment of the programming language Pascal, SIGPLAN Notices, 10, 23-30.

5. Implementation Feasibility

A large number of the people currently involved in the standardisation effort are suppliers of Pascal compilers. It is expected that most (if not all) suppliers will be willing to conform with the standard.

In parallel with the standardisation effort a suite of programs is being developed which should form the basis of a conformance test. This work is being performed jointly by groups based in Australia and the UK.

It is not intended that the standardisation of Pascal will introduce any changes which will cause any currently legal Pascal programs to become illegal. With one or two notable exceptions, the charges envisaged relate to matters of detail. The exceptions are certain points e.g. type compatibility, where it is known that there is currently no generally accepted definition. This lack of a satisfactory definition need not prevent the writing of portable Pascal programs.

6. Closely Related Standards Activities

Those activities which are directed towards the production of a standard for Pascal have already been described. The US Department of Defense is currently engaged in the creation of a new 'real-time' language. (The "Ironman" Project). It has been stated that the resulting language will be Pascal-based. Since Pascal has no 'real-time' features and is consequently not suitable for such applications, there is no conflict of interests. Furthermore, there are several resident microprocessor implementations of Pascal. It is believed that the size of the DOD language would necessitate cross-compilation, if it were to be used on a microprocessor.

7. Outline of the Work Program

The work program has already been started in the form of work towards a British standard for Pascal. It consists of the following steps:

1. The production of a list of the omissions from and the deficiencies of the Pascal Report. This is called the Attention List. (This could be completed by mid-1978).
2. The solution of these problems.
3. The production of a draft proposal, possibly as an edited version of the Pascal Report.

It is not intended that the standardisation effort will involve any development of the language.

# POLICY: PASCAL USER'S GROUP (78/10/01)

Purposes: Pascal User's Group (PUG) tries to promote the use of the programming language Pascal as well as the ideas behind Pascal. PUG members help out by sending information to Pascal News, the most important of which is about implementations (out of the necessity to spread the use of Pascal).

The increasing availability of Pascal makes it a viable alternative for software production and justifies its further use. We all strive to make using Pascal a respectable activity.

Membership: Anyone can join PUG: particularly the Pascal user, teacher, maintainer, implementor, distributor, or just plain fan. Memberships from libraries are also encouraged.

See the ALL-PURPOSE COUPON for details.

FACTS ABOUT Pascal, THE PROGRAMMING LANGUAGE:

Pascal is a small, practical, and general purpose (but not all-purpose) programming language possessing algorithmic and data structures to aid systematic programming. Pascal was intended to be easy to learn and read by humans, and efficient to translate by computers.

Pascal has met these design goals and is being used quite widely and successfully for:

* teaching programming concepts
* developing reliable "production" software
* implementing software efficiently on today's machines
* writing portable software

Pascal is a leading language in computer science today and is being used increasingly in the world's computing industry to save energy and resources and increase productivity.

Pascal implementations exist for more than 62 different computer systems, and the number increases every month. The Implementation Notes section of Pascal News describes how to obtain them.

The standard reference and tutorial manual for Pascal is:

Pascal - User Manual and Report (Second, study edition)

by Kathleen Jensen and Niklaus Wirth

Springer-Verlag Publishers: New York, Heidelberg, Berlin

1978 (corrected printing), 167 pages, paperback, $6.90.

Introductory textbooks about Pascal are described in the Here and There Books section of Pascal News.

The programming language Pascal was named after the mathematician and religious fanatic Blaise Pascal (1623-1662). Pascal is not an acronym.

Pascal User's Group is each individual member's group. We currently have more than 2712 active members in more than 41 countries. This year Pascal News is averaging more than 120 pages per issue.

Return to:

University Computer Center
227 Experimental Engineering Building
208 Southeast Union Street
University of Minnesota
Minneapolis, Minnesota 55455 USA

return postage guaranteed
address correction requested