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


- 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....: 19?-'.

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