Rm 217 Browse Copy

PASCAL USERS GROUP

Pascal News

NUMBER 20

COMMUNICATIONS ABOUT THE PROGRAMMING LANGUAGE PASCAL BY PASCALERS

DECEMBER, 1980

7185.1 Recommended?

Bulk Rate
U.S. Postage
PAID
Atlanta, Ga.
Permit No. 2854

Return to:

PASCAL USERS GROUP
P.O. Box 888524
Atlanta, GA 30338

Return postage guaranteed
Address Correction requested

ATIN: ROOM 217 BROUSE COPY [81]
UNIV. OF MINNESOTA
UCC : 227EX
MINNEAPOLIS, MN 55455

POLICY: PASCAL NEWS (15-Sep-80)

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

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

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

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

* Pascal News is produced 3 or 4 times during a year; usually in March, June, September, and December.

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

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

* Pascal News is divided into flexible sections:

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

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

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

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

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

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

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

- - - - - - ALL-PURPOSE COUPON - - - - - - (15-Sep-80)

Pascal User's Group, c/o Rick Shaw
P.O. Box 888524
Atlanta, Georgia 30338 USA

NOTE

- Membership fee and All Purpose Coupon is sent to your Regional Representative.
- SEE THE POLICY SECTION ON THE REVERSE SIDE FOR PRICES AND ALTERNATE ADDRESS if you are located in the European or Australasian Regions.
- Membership and Renewal are the same price.
- Note the discounts below, for multi-year subscription and renewal.
- The U. S. Postal Service does not forward Pascal News.

- -

		USA	Europe	Aust.
	[ ] 1 year	$10.	£6.	A$ 8.
[ ] Enter me as a new member for:				
	[ ] 2 years	$18.	£10.	A$ 15.
[ ] Renew my subscription for:				
	[ ] 3 years	$25.	£14.	A$ 20.

[ ] Send Back Issue(s) _____

[ ] My new address/phone is listed below

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

[ ] Comments: _____

_____

_____

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

CHECK no. _____

NAME _____

ADDRESS _____

_____

_____

PHONE _____

COMPUTER _____

DATE _____

JOINING PASCAL USER'S GROUP?

- Membership is open to anyone: Particularly the Pascal user, teacher, maintainer, implementor, distributor, or just plain fan.
- Please enclose the proper prepayment (check payable to "Pascal User's Group"); we will not bill you.
- Please do not send us purchase orders; we cannot endure the paper work!
- When you join PUG any time within a year: January 1 to December 31, you will receive all issues of Pascal News for that year.
- We produce Pascal News as a means toward the end of promoting Pascal and communicating news of events surrounding Pascal to persons interested in Pascal. We are simply interested in the news ourselves and prefer to share it through Pascal News. We desire to minimize paperwork, because we have other work to do.

- American Region (North and South America): Send $10.00 per year to the address on the reverse side. International telephone: 1-404-252-2600.
- European Region (Europe, North Africa, Western and Central Asia): Join through PUG (UK). Send £5.00 per year to: Pascal Users Group, c/o Computer Studies Group, Mathematics Department, The University, Southampton SO9 5NH, United Kingdom; or pay by direct transfer into our Post Giro account (28 513 4000); International telephone: 44-703-559122 x700.
- Australasian Region (Australia, East Asia - incl. Japan): PUG(AUS). Send $A10.00 per year to: Pascal Users Group, c/o Arthur Sale, Department of Information Science, University of Tasmania, Box 252C GPO, Hobart, Tasmania 7001, Australia. International telephone: 61-02-23 0561 x435

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

RENEWING?

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

ORDERING BACK ISSUES OR EXTRA ISSUES?

- Our unusual policy of automatically sending all issues of Pascal News to anyone who joins within a year means that we eliminate many requests for backissues ahead of time, and we don't have to reprint important information in every issue--especially about Pascal implementations!
- Issues 1 .. 8 (January, 1974 - May 1977) are out of print. (A few copies of issue 8 remain at PUG(UK) available for £2 each.)
- Issues 9 .. 12 (September, 1977 - June, 1978) are available from PUG(USA) all for $15.00 and from PUG(AUS) all for $A15.00
- Issues 13 .. 16 are available from PUG(UK) all for £10; from PUG(AUS) all for $A15.00; and from PUG(USA) all for $15.00.
- Extra single copies of new issues (current academic year) are: $5.00 each - PUG(USA); £3 each - PUG(UK); and $A5.00 each - PUG(AUS).

SENDING MATERIAL FOR PUBLICATION?

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

Editor's Contribution

RENEWING

This is the last issue of the year. (Bet you thought it would never get here!!) So if you have not renewed yet, RENEW NOW !!! It is easy to tell if you need to renew, because all you have to do is look at your mailing label. (Except in the Australasian Region.) If the number in square brackets says 80 (ie. " [80] ") then this is your last issue. This number is the year your subscription expires.

THIS ISSUE

This issue contains the full text of the "Second Draft" of the proposed ISO Pascal Standard. I hope it is the last one we publish; because it is the last one! Andy Mickel (remember Andy?!) was present at the X3J9 meeting in Huntsville, and has also been doing plenty of long distance politicing for this standard. He asked if he could write a guest editorial and the text follows.

Rick

UNIVERSITY OF MINNESOTA
TWIN CITIES

University Computer Center
227 Experimental Engineering Building
208 Union Street S.E.
Minneapolis, Minnesota 55455

1981-01-08

This special issue of Pascal News presents the second draft proposal of the ISO Pascal Standard now out for public comment and voting by the appropriate national bodies. More formally this document is known as (revised) DP7185.1.

[Alice Droogan, ISO TC97/SC5 Secretariat said to send all comment to: Joint Pascal Committee, c/o Larry B. Weber, IBM, General Products Division, 555 Bailey Avenue, San Jose, CA 95150 USA. See also bottom of page 69, Pascal News #18]

As was reported by Jim Miner on page 74 of Pascal News #19, the first draft received 11 yes and 4 no votes. Most of the people I know associated with ISO Pascal Standards activities (including myself) expect unanimous approval on this draft. There are several things I can say about this:

1. The ISO Pascal standard is badly needed now and is overdue, but it will have set speed records in approval.
2. Even though the draft standard is imperfect (and always will be) the realization among those experts from the ISO Working Group is that extra time spent on the draft in an effort to perfect it has reached the point of diminishing returns.
3. This draft can be expected to be very close to the final standard.
4. Pascal users will at last benefit from a single standard when it will be adopted by the national standards groups in ISO member countries (such as in the USA by ANSI/IEEE/NBS and the Federal Govt.).

What I wrote two years ago in an editorial in Pascal News #14 which introduced the third BSI working draft of a Pascal Standard still applies:

> Pascal Standards should be given special consideration (in other words, there are not necessarily applicable precedents found in the standards processes of other languages). The Pascal Standards process has been a model phenomenon in Computer Science history.
>
> First and foremost Pascal was designed by a single person (Niklaus Wirth) and is not a committee-designed language. Pascal Standards Committees have so far rightly refrained from adding committee-designed features. Secondly, Pascal is the first major programming language standardized outside the United States. As I've said before, it has European origins but to be more accurate, Pascal is truly international. I think that's wonderful and neat!

Pascal is in very wide use (even though there are dozens of programmers ignorant of its impact and uses). Its design goals mentioned in my Pascal News #14 editorial have been met and exceeded (even though there are plenty of computing people who deny this).

Finally, let me reiterate the implications of an imperfect Pascal standard. In the time given, with the people involved, and with the resources we've had, it's a remarkable achievement. (Thank you, Tony Addyman!) And it is still imperfect. But now the existence of a finished standard is more important thatn spending any more time.

In spite of the attitude of many of us technical people, you can't always fix certain things--technical problems don't always have clean solutions. It's not clear in some cases that solutions can be attained. In other words, if you put enough constraints on a problem, it could be the case that the set of solutions is empty.

Therefore, regarding the conformant-array feature I am happy; after having listened to the large volume of discussion, I know that it is equivalent in quality to any alternative. To repeat a familiar refrain, if there had been a natural solution, Niklaus would have incorporated it in the first place.

He's said so himself.

-J.J.

PASCAL NEWS #20 DECEMBER, 1980

american national standards institute, inc · 1430 broadway, new york, n.y. 10018 · (212) 354-3300

Cable: Standards, New York International Telex: 42 42 96 ANSI UI

January 21, 1981

Dear Mr. Shaw:

Enclosed please find second draft proposal ISO/DP 7185 - Specification for the Computer Programming Language - Pascal. This document is being circulated to 97/5 committee members for voting on by March 31, 1981.

Comments on the document are welcome and will be considered but must be in written form and must be received by March 31, 1981. Please address all comments to ANSI's X3J9 Chairman:

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

Comments should be clearly marked with the name, address and telephone number of the commentor, the section and subsection to which the comment applies, and a rationale or explanation for any proposed text changes. Specific proposed text changes are the most desirable form for comments, but general changes or criticisms, or questions, are also welcome.

Sincerely,

Alice Droogan

Alice Droogan
Secretariat ISO/TC 97/SC 5

AD/MAC
Encl.

ISO/TC 97/SC 5 N 595
January 1981

DP7185 SPECIFICATION FOR THE COMPUTER PROGRAMMING LANGUAGE Pascal

CONTENTS

Foreword

The language Pascal was designed by Professor Niklaus Wirth to satisfy two principal aims:
(a) to make available a language suitable for teaching programming as a systematic discipline based on certain fundamental concepts clearly and naturally reflected by the language.
(b) to define a language whose implementations could be both reliable and efficient on then available computers.

Second Draft Proposal

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

In drafting this standard the continued stability of Pascal has been a prime objective. However, apart from changes to clarify the specification, two major changes have been introduced:
(a) the syntax used to specify procedural and functional parameters has been changed to require the use of a procedure or function heading, as appropriate (see 6.6.3.1). This change was introduced to overcome a language insecurity;
(b) a fifth kind of parameter, the conformant array parameter, has been introduced (see 6.6.3.7). With this kind of parameter, the required bounds of the index-type of an actual parameter are not fixed, but are restricted to a specified range of values.

0. INTRODUCTION

The appendices are included for the convenience of the reader of this standard. They do not form a part of the requirements of this standard.

1. SCOPE OF THIS STANDARD

1.1 This standard specifies the semantics and syntax of the computer programming language Pascal by specifying requirements for a processor and for a conforming program. Two levels of compliance are defined for both processors and programs.

1.2 This standard does not specify
(a) the size or complexity of a program and its data that will exceed the capacity of any specific data processing system or the capacity of a particular processor;
(b) the minimal requirements of a data processing system that is capable of supporting an implementation of a processor for Pascal;
(c) the method of activating the program-block or the set of commands used to control the environment in which a Pascal program is transformed and executed;
(d) the mechanism by which programs written in Pascal are transformed for use by a data processing system;
(e) the method for reporting errors or warnings;
(f) the typographical representation of a program published for human reading.

2. REFERENCES

None.

3. DEFINITIONS

3.1 error. A violation by a program of the requirements of this standard whose detection by a processor is optional.
3.2 implementation-defined. Possibly differing between processors, but defined for any particular processor.
3.3 implementation-dependent. Possibly differing between processors and not necessarily defined for any particular processor.
3.4 processor. A compiler, interpreter, or other mechanism which accepts the program as input and either executes it, prepares it for execution, or both.

4. DEFINITIONAL CONVENTIONS

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

Table 1. Metalanguage symbols

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

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

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

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

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

> the x of a y: refers to the x occurring directly in a production defining y.
>
> the x in a y: is synonymous with "the x of a y".
>
> a y containing an x: refers to any x directly or indirectly derived from y.
>
> the y closest-containing an x: that y which contains an x but does not contain another y containing that x.

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

5. COMPLIANCE

> NOTE. There are two levels of compliance - level 0 and level 1. Level 0 does not include conformant array parameters. Level 1 does include conformant array parameters.

5.1 Processors
A processor complying with the requirements of this standard shall:
(a) if it complies at level 0, accept all the features of the language specified in clause 6, except for 6.6.3.6(e), 6.6.3.7 and 6.6.3.8, with the meanings defined in clause 6;
(b) if it complies at level 1, accept all the features of the language specified in clause 6 with the meanings defined in clause 6;
(c) not require the inclusion of substitute or additional language elements in a program in order to accomplish a feature of the language that is specified in clause 6;
(d) be accompanied by a document that provides a definition of all implementation-defined features;
(e) detect any violation by a program of the requirements of this standard that is not designated an error;
(f) treat each violation that is designated an error in at least one of the following ways:

Second Draft Proposal

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

(g) be accompanied by a document that separately describes any features accepted by the processor that are not specified in clause 6. Such extensions shall be described as being 'extensions to Pascal specified by ISO7185: 198-'.
(h) be able to process in a manner similar to that specified for errors any use of any such extension;
(i) be able to process in a manner similar to that specified for errors any use of an implementation-dependent feature.

5.2 Programs

A program complying with the requirements of this standard shall:
(a) if it complies at level 0, use only those features of the language specified in clause 6, except for 6.6.3.6(e), 6.6.3.7 and 6.6.3.8;
(b) if it complies at level 1, use only those features of the language specified in clause 6;
(c) not rely on any particular interpretation of implementation-dependent features.

NOTE. The results produced by the processing of a complying program by different complying processors are not required to be the same.

6. REQUIREMENTS

6.1 Lexical tokens

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

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

```
letter = "a"|"b"|"c"|"d"|"e"|"f"|"g"|"h"|"i"|"j"|"k"|"l"|"m"|
         "n"|"o"|"p"|"q"|"r"|"s"|"t"|"u"|"v"|"w"|"x"|"y"|"z" .

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

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

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

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

6.1.3 Identifiers. Identifiers may be of any length. All characters of an identifier shall be significant. No identifier shall have the same spelling as any word-symbol.

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

Examples:

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

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

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

NOTE. On many processors the directive external is used to specify that the procedure-block or function-block corresponding to that procedure-heading or function-heading is external to the program-block. Usually it is in a library in a form to be input

to, or that has been produced by, the processor.

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

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

Examples:

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

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

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

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

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

Examples:

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


6.1.8 Token separators. The construct

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

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

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

6.2 Blocks, scope and activations
6.2.1 Block. A block closest-containing a label-declaration-part in which a label occurs shall closest-contain exactly one statement in which that label occurs. The occurrence of a label in the label-declaration-part of a block shall be its defining-point as a label for the region which is the block.

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

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

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

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

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

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

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

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

All variables contained by an activation, except for those listed as program-parameters, shall be totally-undefined at the commencement of that activation.

6.2.2 Scope
6.2.2.1 Each identifier or label contained by the program-block shall have a defining-point.
6.2.2.2 Each defining-point shall have a region that is a part of the program text, and a scope that is a part or all of that region.
6.2.2.3 The region of each defining-point is defined elsewhere (see 6.2.1, 6.2.2.10, 6.3, 6.4.1, 6.4.2.3, 6.4.3.3, 6.5.1, 6.5.3.3, 6.6.1, 6.6.2, 6.6.3.1, 6.8.3.10).
6.2.2.4 The scope of each defining-point shall be its region (including all regions enclosed by that region) subject to 6.2.2.5 and 6.2.2.6.
6.2.2.5 When an identifier or label that has a defining-point for region A has a further defining-point for some region B enclosed by A, then region B and all regions enclosed by B shall be excluded from the scope of, the defining-point for region A.
6.2.2.6 The field-identifier of the field-specifier of a field-designator (see 6.5.3.3) shall be one of the field-identifiers associated with a component of the record-type possessed by the record-variable of the field-designator.

Second Draft Proposal

6.2.2.7 The scope of a defining-point of an identifier or label shall include no other defining-point of the same identifier or label.
6.2.2.8 Within the scope of a defining-point of an identifier or label, all occurrences of that identifier or label shall be designated applied occurrences, except for an occurrence that constituted the defining-point of that identifier or label; such an occurrence shall be designated a defining occurrence. No occurrence outside that scope shall be an applied occurrence.
6.2.2.9 The defining-point of an identifier or label shall precede all applied occurrences of that identifier or label contained by the program-block with one exception, namely that a type-identifier may have an applied occurrence in the domain-type of any new-pointer-types contained by the type-definition-part that contains the defining-point of the type-identifier.
6.2.2.10 Identifiers that denote required constants, types, procedures and functions shall be used as if their defining-points have a region enclosing the program.
6.2.2.11 Whatever an identifier or label denotes at its defining-point shall be denoted at all applied occurrences of that identifier or label.

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

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

6.2.3.3. The activation of a procedure or function shall be the activation of the block of its procedure-block or function-block, respectively, and shall be designated within:
(a) the activation containing the procedure or function; and
(b) all activations that that containing activation is within.

Second Draft Proposal

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

Within an activation, an applied occurrence of a label or variable-identifier, or of a procedure-identifier or function-identifier local to the block of the activation, shall denote the corresponding program-point, variable, procedure, or function, respectively, of that activation.

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

6.2.3.5. The algorithm, program-points, variables, procedures and functions, if any, shall exist until the termination of the activation.

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

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

The occurrence of an identifier in a constant-definition of a constant-definition-part of a block shall constitute its defining-point for the region that is the block. The constant shall not contain an applied occurrence of the identifier in the constant-definition. Each applied occurrence of that identifier shall be a constant-identifier and shall denote the value denoted by the constant of the constant-definition. A constant-identifier in a constant containing an occurrence of a sign shall have been defined to denote a value of real-type or of integer-type.

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

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

The occurrence of an identifier in a type-definition of a type-definition-part of a block shall constitute its defining-point for the region that is the block. Each applied occurrence of that identifier shall be a type-identifier and shall denote the same type as that which is denoted by its type-denoter. Except for applied occurrences as the domain-type of a new-pointer-type, the type-denoter shall not contain an applied occurrence of the identifier in the type-definition.

Types shall be classified as simple, structured or pointer types. The required types shall be denoted by predefined type-identifiers (see 6.4.2.2 and 6.4.3.5).

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

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

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

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

6.4.2.2 Required simple-types. The following types shall exist:

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

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

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

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

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

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

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

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

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

6.4.2.3 Enumerated-types. An enumerated-type shall determine an ordered set of values by enumeration of the identifiers that denote those values. The ordering of these values shall be determined by the sequence in which their identifiers are enumerated, i.e. if x precedes y then x is less than y. The ordinal number of a value that is of an enumerated-type shall be determined by mapping all the values of the type as their identifiers occur in the identifier-list of the enumerated-type on to consecutive non-negative values of integer-type starting from zero.

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

The occurrence of an identifier in the identifier-list of an

enumerated-type shall constitute its defining-point as a constant-identifier for the region which is the block closest-containing the enumerated-type.

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

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

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

Examples:
```
1..100
-10..+10
red..green
'0'..'9'
```

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

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

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

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

NOTE. The ways in which the treatment of entities of a type is affected by whether or not the type is designated packed are specified in 6.4.3.2, 6.4.5, 6.6.3.3, 6.6.3.8, 6.6.5.4 and 6.7.1.

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

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

Examples:

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

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

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

Example 1.
```
array[Boolean] of array[1..10] of array[size] of real
array[Boolean] of array[1..10,size] of real
array[Boolean,1..10,size] of real
array[Boolean,1..10] of array[size] of real
```
Example 2.
```
packed array[1..10,1..8] of Boolean
packed array[1..10] of packed array[1..8] of Boolean
```

Let i denote a value of the index-type; let v[i] denote a value of that component of the array-type that corresponds to the value i by the structure of the array-type; let the smallest and largest values specified by the index-type be denoted by m and n; and let k = (ord(n)-ord(m)+1) denote the number of values specified by the index-type. Then the values of the array-type shall be the distinct k-tuples of the form:
```
(v[m], ... ,v[n])
```

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

of the array-type is c raised to the power k.

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

The correspondence of character-strings to values of string-types is obtained by relating the individual characters of the character-string, taken in left to right order, to the components of the values of the string-type in order of increasing index.

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

6.4.3.3 Record-types. The structure and values of a record-type shall be the structure and values of the field-list of the record-type.

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

A field-list which contains neither a fixed-part nor a variant-part shall have no components, shall define a single null value, and shall be designated empty.

The occurrence of an identifier in the identifier-list of a record-section of a fixed-part of a field-list shall constitute its defining-point as a field-identifier for the region which is the record-type closest-containing the field-list, and shall associate the field-identifier with a distinct component, which shall be designated a field, of the record-type and of the field-list. That component shall have the type denoted by the type-denoter of the record-section.

The field-list closest-containing a variant-part shall have a distinct component which shall have the values and structure defined by the variant-part.

Let Vi denote the value of the i-th component of a non-empty field-list having m components; then the values of the field-list shall be distinct m-tuples of the form

(V1, V2, ..., Vm).

NOTE. If the type of the i-th component has Fi values, then the cardinality of the set of values of the field-list shall be (F1 * F2 * ... * Fm).

A tag-type shall denote the type denoted by the ordinal-type-identifier of the tag-type. A case-constant shall denote the value denoted by the constant of the case-constant.

The type of each case-constant in the case-constant-list of a variant of a variant-part shall be compatible with the tag-type of the variant-selector of the variant-part. The values denoted by all case-constants of a type that is required to be compatible with a given tag-type shall be distinct and the set thereof shall be equal to the set of values specified by the tag-type. The values denoted by the case-constants of the case-constant-list of a variant shall be designated as corresponding to the variant.

With each variant-part shall be associated a type designated the selector-type possessed by the variant-part. If the variant-selector of the variant-part contains a tag-field, or if the case-constant-list of each variant of the variant-part contains only one case-constant, then the selector-type shall be denoted by the tag-type, and each variant of the variant-part shall be associated with those values specified by the selector-type denoted by the case-constants of the case-constant-list of the variant. Otherwise, the selector-type possessed by the variant-part shall be a new ordinal-type constructed such that there is exactly one value of the type for each variant of the variant-part, and no others, and each variant shall be associated with a distinct value of that type.

Each variant-part shall have a component which shall be designated the selector of the variant-part, and which shall possess the selector-type of the variant-part. If the variant-selector of the variant-part contains a tag-field, then the occurrence of an identifier in the tag-field shall constitute the defining-point of the identifier as a field-identifier for the region which is the record-type closest-containing the variant-part, and shall associate the field-identifier with the selector of the variant-part. The selector shall be designated a field of the record-type if and only if it is associated with a field-identifier.

Each variant of a variant-part shall denote a distinct component of the variant-part; the component shall have the values and structure of the field-list of the variant, and shall be associated with those values specified by the selector-type possessed by the variant-part which are associated with the variant. The value of the selector of the variant-part shall cause the associated variant and component of the variant-part to be in a state that shall be designated active. The values of a variant-part shall be the distinct pairs

(k, Xk)'

where k represents a value of the selector of the variant-part, and Xk is a value of the field-list of the active variant of the variant-part.

NOTES

1. If there are n values specified by the selector-type, and if the field-list of the variant associated with the i-th value has Ti values, then the cardinality of the set of values of the variant-part is (T1 + T2 + ... + Tn). There is no component of a value of a variant-part corresponding to any non-active variant of the variant-part.

2. Restrictions placed on the use of fields of a record-variable pertaining to variant-parts are specified in 6.5.3.3, 6.6.3.3 and 6.6.5.3.

Examples:

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

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

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

6.4.3.4 Set-types. A set-type shall determine the set of values that is structured as the powerset of its base-type. Thus each value of a set-type shall be a set whose members shall be unique values of the base-type.

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

base-type = ordinal-type

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

Examples:
set of char
set of (club, diamond, heart, spade)

NOTE. If the base-type of a set-type has b values then the cardinality of the set of values is 2 raised to the power b.

For every ordinal-type S, there exists an unpacked set designated the unpacked canonical set-of-T type and there exists a packed set type designated the packed canonical set-of-T type. If S is a subrange-type then T is the host type of S; otherwise T is S. Each value of the type set of S is also a value of the unpacked canonical set-of-T type, and each value of the type packed set of S is also a value of the packed canonical set-of-T type.

6.4.3.5 File-types.

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

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

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

Examples:
file of real
file of vector

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

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

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

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

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

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

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

A file-type also shall define implicitly a type designated a mode-type having exactly two values which are designated Inspection and Generation.

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

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

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

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

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

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

There shall be a file-type that is denoted by the required structured-type-identifier text. The structure of the type denoted by text shall define an additional sequence-type whose values shall be designated lines. A line shall be a sequence x~S(e), where x is a

sequence of components having the char-type, and e represents a special component value, which shall be designated an end-of-line, and which shall be indistinguishable from the char value space except by the required function eoln (6.6.6.5) and by the required procedures reset (6.6.5.2), writeln (6.9.5), and page (6.9.6). If x is a line then no component of x other than x.last shall be an end-of-line. This definition shall not be construed to determine the underlying representation, if any, of an end-of-line component used by a processor.

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

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

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

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

NOTE. In rule (b), y may be considered, especially if it is non-empty, to be a partial line which is being generated. Such a partial line cannot occur during inspection of a file. Also, y does not correspond to t.R since t.R is the empty sequence if t.M = Generation.

A variable that possesses the type denoted by the required structured-type-identifier text shall be designated a textfile.

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

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

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

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

```
pointer-type = new-pointer-type | pointer-type-identifier .
new-pointer-type = "^" domain-type .
domain-type = type-identifier .
```


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

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

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

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

6.4.7 Example of a type-definition-part

```
type
  natural = 0..maxint;
  count = integer;
  range = integer;
  colour = (red, yellow, green, blue);
  sex = (male, female);
  year = 1900..1999;
  shape = (triangle, rectangle, circle);
  punchedcard = array[1..80] of char;
  charsequence = file of char;
  polar = record
            r : real;
            theta : angle
          end;
```