PASCAL USERS GROUP

Pascal News

NUMBER 20

COMMUNICATIONS ABOUT THE PROGRAMMING LANGUAGE PASCAL BY PASCALERS

DECEMBER, 1980

7185.1

Recommended?

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.
[ ] Enter me as a new member for:	[ ] 1 year	$10.	£6.	A$ 8.
	[ ] 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.

– Andy

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.

Second Draft Proposal

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 orisinal 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

Second Draft Proposal

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:

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'
```

Second Draft Proposal

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

Second Draft Proposal

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.

Second Draft Proposal

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.

Second Draft Proposal

(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;
```

Second Draft Proposal

```
indextype = 1..limit;
vector = array [indextype] of real;
person = ^persondetails;
persondetails =
        record
          name, firstname : charsequence;
          age : integer;
          married : Boolean;
          father, child, siblins : person;
          case s : sex of
            male :
              (enlisted,bearded : Boolean);
            female :
              (mother,programmer : Boolean)
        end;
FileOfInteger = file of integer;
```

NOTES

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

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

6.5 Declarations and denotations of variables
6.5.1 Variable-declarations. A variable is an entity to which a (current) value may be attributed (see 6.8.2.2). Each identifier in the identifier-list of a variable-declaration shall denote a distinct variable possessing the type denoted by the type-denoter of the variable-declaration.

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

The occurrence of an identifier in the identifier-list of a variable-declaration of the variable-declaration-part of a block shall constitute its defining-point as a variable-identifier for the region that is the block. The structure of a variable possessing a structured-type shall be the structure of the structured-type. A use of a variable-access shall be an access, at the time of the use, to the variable thereby denoted. A variable-access, according to whether it is an entire-variable, a component-variable, an identified-variable, or a buffer-variable, shall denote either a declared variable, or a component of a variable, a variable which is identified by a pointer value (see 6.4.4), or a buffer-variable, respectively.

variable-access = entire-variable | component-variable |
identified-variable | buffer-variable .

An assigning-reference to a variable shall occur if any of the six statements that follow is true.
(a) The variable is denoted by the variable-access of an assignment-statement.
(b) The variable is denoted by an actual variable parameter in a function-designator or procedure-statement.
(c) The variable is denoted by an actual parameter in a procedure-statement that specifies the activation of the required procedure read or the required procedure readln.
(d) The variable occurs as the control-variable of a for-statement.
(e) A procedure-statement or a function-designator contains a procedure-identifier associated with a procedure-block containing an assigning-reference to the variable.
(f) A procedure-statement or a function-designator contains a function-identifier associated with a function-block containing an assigning-reference to the variable.

Example of a variable-declaration-part

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

Second Draft Proposal

```
date : record
         month : 1..12;
         year : integer
       end;
```

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

6.5.2 Entire-variables.

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

6.5.3 Component-variables

6.5.3.1 General. A component of a variable shall be a variable. A component-variable shall denote a component of a variable. A reference, assigning-reference or access to a component of a variable shall constitute a reference, assigning-reference or access, respectively, to the variable. The value, if any, of the component of a variable shall be the same component of the value, if any, of the variable.

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

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

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

An array-variable shall be a variable-access that denotes a variable possessing an array-type. For an indexed-variable closest-containing a single index-expression, the value of the index-expression shall be assignment-compatible with the index-type of the array-type. The component denoted by the indexed-variable shall be the component that corresponds to the value of the index-expression by the mapping of the type possessed by the array-variable (see 6.4.3.2).

Examples:

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

If the array-variable is itself an indexed-variable an abbreviation may be used. In the abbreviated form, a single comma shall replace the sequence "]" "[" that occurs in the full form. The abbreviated form and the full form shall be equivalent.

Second Draft Proposal

Examples:

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

NOTE. The two examples denote the same component variable.

6.5.3.3 Field-designators. A field-designator either shall denote that component of the record-variable of the field-designator which is associated with the field-identifier of the field-specifier of the field-designator, by the record-type possessed by the record-variable; or shall denote the variable denoted by the field-designator-identifier (see 6.8.3.10) of the field-designator. A record-variable shall be a variable-access that denotes a variable possessing a record-type.

The occurrence of a record-variable in a field-designator shall constitute the defining-point of the field-identifiers associated with components of the record-type possessed by the record-variable, for the region that is the field-specifier of the field-designator.

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

Examples:

```
p2^.mother
coord.theta
```

An access to a component of a variant of a variant-part, where the selector of the variant-part is not a field, shall attribute to the selector that value specified by its type which is associated with the variant.

It shall be an error unless a variant is active for the entirety of each reference and access to each component of the variant.

When a variant becomes not active, all of its components shall become totally-undefined.

NOTE. If the selector of a variant-part is undefined, then no variant of the variant-part is active.

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

```
identified-variable = pointer-variable "^" .
pointer-variable = variable-access .
```

A variable created by the required procedure new (see 6.6.5.3) shall

be accessible until the termination of the activation of the program-block or until the variable is made inaccessible (see the required procedure dispose, 6.6.5.3).

NOTE. The accessibility of the variable also depends on the existence of a pointer-variable which has attributed to it the corresponding identifying value.

A pointer-variable shall be a variable-access that denotes a variable possessing a pointer-type. It shall be an error if the pointer-variable of an identified-variable either denotes a nil-value or is undefined. It shall be an error to remove from its pointer-type the identifying-value of an identified variable (see 6.6.5.3) when a reference to the identified variable exists.

Examples:
```
p1^
p1^.father^
p1^.siblings^.father^
```

6.5.5 Buffer-variables. A file-variable shall be a variable-access that denotes a variable possessing a file-type. A buffer-variable shall denote a variable associated with the variable denoted by the file-variable of the buffer-variable. A buffer-variable associated with a textfile shall possess the char-type; otherwise, a buffer-variable shall possess the component-type of the file-type possessed by the file-variable of the buffer-variable.

```
buffer-variable = file-variable "^" .
file-variable = variable-access .
```

Examples:
```
input^
pooltape[2]^
```

It shall be an error to alter the value of a file-variable f when a reference to the buffer-variable f^ exists. A reference or access to a buffer-variable shall constitute a reference or access, respectively, to the associated file-variable.

6.6 Procedure and function declarations

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

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

The occurrence of a formal-parameter-list in a procedure-heading of a procedure-declaration shall define the formal parameters of the procedure-block, if any, associated with the identifier of the procedure-heading to be those of the formal-parameter-list.

The occurrence of an identifier in the procedure-heading of a procedure-declaration shall constitute its defining-point as a procedure-identifier for the region that is the block closest-containing the the procedure-declaration.

Each identifier having a defining-point as a procedure-identifier in a procedure-heading of a procedure-declaration closest-containing the directive "forward" shall have exactly one of its corresponding occurrences in a procedure-identification of a procedure-declaration, and that shall be in the same procedure-and-function-declaration-part.

The occurrence of a procedure-block in a procedure-declaration associates the procedure-block with the identifier in the procedure-heading, or with the procedure-identifier in the procedure-identification, of the procedure-declaration.

Example of a procedure-and-function-declaration-part:

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

Second Draft Proposal

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

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

Second Draft Proposal

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

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

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

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

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

The occurrence of a formal-parameter-list in a function-heading of a function-declaration shall define the formal parameters of the function-block, if any, associated with the identifier of the function-heading to be those of the formal-parameter-list. The function-block shall contain at least one assignment-statement that attributes a value to the function-identifier (see 6.8.2.2). The value of the function shall be the last value attributed to the function-identifier. It shall be an error if the function is undefined upon completion of the algorithm of an activation of the function-block.

Second Draft Proposal

The occurrence of an identifier in the function-heading of a function-declaration shall constitute its defining-point as a function-identifier possessing the type denoted by the result-type for the region that is the block closest-containing the the function-declaration.

Each identifier having a defining-point as a function-identifier in the function-heading of a function-declaration closest-containing the directive "forward" shall have exactly one of its corresponding occurrences in a function-identification of a function-declaration, and that shall be in the same procedure-and-function-declaration-part.

The occurrence of a function-block in a function-declaration associates the function-block with the identifier in the function-heading, or with the function-identifier in the function-identification, of the function-declaration.

Example of a procedure-and-function-declaration-part

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

Second Draft Proposal

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


function GCD(m,n: natural): natural;
begin
    if n=0 then GCD := m else GCD := GCD(n,m mod n);
end;

{This  example of the use of forward demonstrates how mutual recursion
is helpful in reading a parenthesized expression and converting it  to
some internal form}
function ReadOperand : formula; forward;

function ReadExpression : formula;
  var
    this : formula;
  begin
  this := ReadOperand;
  while IsOperator(nextsym) do
    this := MakeFormula(this, ReadOperator, ReadOperand);
  ReadExpression := this
  end;

function ReadOperand { : formula };
  begin
  if IsOpen(nextsym) then
    begin
    SkipSymbol;
    ReadOperand := ReadExpression;
    {nextsym should be a close}
    SkipSymbol
    end
  else ReadOperand := ReadElement
  end;
```

6.6.3 Parameters

6.6.3.1 General. The identifier-list in a value-parameter-specification shall be a list of value parameters. The

Second Draft Proposal

identifier-list in a variable-parameter-specification shall be a list of variable parameters.

```
formal-parameter-list =
        "(" formal-parameter-section
        {";" formal-parameter-section} ")" .
formal-parameter-section >
        value-parameter-specification |
        variable-parameter-specification |
        procedural-parameter-specification |
        functional-parameter-specification .
```

NOTE. There is also a syntax rule for formal-parameter-section in 6.6.3.7

```
value-parameter-specification =
        identifier-list ":" type-identifier .
variable-parameter-specification =
        "var" identifier-list ":" type-identifier .
procedural-parameter-specification =
        procedure-heading .
functional-parameter-specification =
        function-heading .
```

An identifier that is defined to be a parameter-identifier for the region which is the formal-parameter-list of a procedure-heading shall be designated a formal parameter of the block of the procedure-block, if any, associated with the identifier of the procedure-heading. An identifier that is defined to be a parameter-identifier for the region which is the formal-parameter-list of a function-heading shall be designated a formal parameter of the block of the function-block, if any, associated with the identifier of the function-heading.

The occurrence of an identifier in the identifier-list of a value-parameter-specification or a variable-parameter-specification shall constitute its defining-point as a parameter-identifier for the region that is the formal-parameter-list closest-containing it and its defining-point as the associated variable-identifier for the region that is the block, if any, of which it is a formal parameter.

The occurrence of the identifier of a procedure-heading in a procedural-parameter-specification shall constitute its defining-point as a parameter-identifier for the region that is the formal-parameter-list closest-containing it and its defining-point as the associated procedure-identifier for the region that is the block, if any, of which it is a formal parameter.

The occurrence of the identifier of a function-heading in a functional-parameter-specification shall constitute its defining-point as a parameter-identifier for the region that is the formal-parameter-list closest-containing it and its defining-point as the associated function-identifier for the region that is the block, if any, of which it is a formal parameter.

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

6.6.3.2 Value parameters. The formal parameter and its associated variable-identifier shall denote the same variable. The formal parameter shall possess the type denoted by the type-identifier of the value-parameter-specification. The actual-parameter (see 6.7.3 and 6.8.2.3) shall be an expression whose value is assignment-compatible with the type possessed by the formal parameter. The current value of the expression shall be attributed upon activation of the block to the variable that is denoted by the formal parameter.

6.6.3.3 Variable parameters. The actual-parameter shall be a variable-access. The actual-parameters (see 6.7.3 and 6.8.2.3) corresponding to formal parameters that occur in a single variable-parameter-specification shall all possess the same type. The type possessed by the actual-parameters shall be the same as that denoted by the type-identifier, and the formal parameters shall also possess that type. The actual-parameter shall be accessed before the activation of the block, and this access shall establish a reference to the variable thereby accessed during the entire activation of the block; the corresponding formal parameter and its associated variable-identifier shall denote the referenced variable during the activation.

An actual variable parameter shall not denote a field which is the selector of a variant-part. An actual variable parameter shall not denote a component of a variable that possesses a type that is designated packed.

6.6.3.4 Procedural parameters. The actual-parameter (see 6.7.3 and 6.8.2.3) shall be a procedure-identifier that has a defining-point contained by the program-block. The procedure denoted by the actual-parameter and the procedure denoted by the formal parameter shall have congruous formal-parameter-lists (see 6.6.3.6) if either has a formal-parameter-list. The formal parameter and its associated procedure-identifier shall denote the actual parameter during the entire activation of the block.

6.6.3.5 Functional parameters. The actual-parameter (see 6.7.3 and 6.8.2.3) shall be a function-identifier that has a defining-point contained by the program-block. The function denoted by the actual-parameter and the function denoted by the formal parameter shall have the same result-type and shall have congruous formal-parameter-lists (see 6.6.3.6) if either has a formal-parameter-list. The formal parameter and its associated function-identifier shall denote the actual parameter during the entire activation of the block.

6.6.3.6 Parameter list congruity. Two formal-parameter-lists shall be

congruous if they contain the same number of formal-parameter-sections and if the formal-parameter-sections in corresponding positions match. Two formal-parameter-sections shall match if any of the statements that follow is true.

(a) They are both value-parameter-specifications containing the same number of parameters and the type-identifier in each value-parameter-specification denotes the same type.

(b) They are both variable-parameter-specifications containing the same number of parameters and the type-identifier in each variable-parameter-specification denotes the same type.

(c) They are both procedural-parameter-specifications and the formal-parameter-lists of the procedure-headings thereof are congruous.

(d) They are both functional-parameter-specifications, the formal-parameter-lists of the function-headings thereof are congruous, and the type-identifiers of the result-types of the function-headings thereof denote the same type.

(e) They are both conformant-array-parameter-specifications containing the same number of parameters and equivalent conformant-array-schemas. Two conformant-array-schemas shall be equivalent if all of the four statements which follow are true.
(1) There is a single index-type-specifiecation in each conformant-array-schema.
(2) The ordinal-type-identifier in each index-type-specification denotes the same type.
(3) Either the (component) conformant-array-schemas of the conformant-array-schemas are equivalent or the type-identifiers of the conformant-array-schemas denote the same type.
(4) Either both conformant-array-schemas are packed-conformant-array-schemas or both are unpacked-conformant-array-schemas.

NOTES
1. The abbreviated conformant-array-schema and its corresponding full form are equivalent (see 6.6.3.7)

2. The contents of (e) above do not apply to level 0.

6.6.3.7 Conformant array parameters.

NOTE. This clause does not apply to level 0.

The occurrence of an identifier in the identifier-list of a conformant-array-parameter-specification shall constitute its defining-point as a parameter-identifier for the region that is the formal-parameter-list closest-containing it and its defining-point as the associated variable-identifier for the region that is the block, if any, of which it is a formal parameter.

The occurrence of an identifier in an index-type-specification shall constitute its defining-point as a bound-identifier for the region that is the formal-parameter-list closest-containing it and for the region that is the block, if any, whose formal parameters are

Second Draft Proposal

specified by that formal-parameter-list.

```
  formal-parameter-section >
        conformant-array-parameter-specification .
  conformant-array-parameter-specification =
        "var" identifier-list ":" conformant-array-schema .
  conformant-array-schema =
        (packed-conformant-array-schema |
        unpacked-conformant-array-schema) .
  packed-conformant-array-schema =
        "packed" "array" "[" index-type-specification "]"
        "of" type-identifier .
  unpacked-conformant-array-schema =
        "array" "[" index-type-specification
        { ";" index-type-specification } "]" "of"
        ( type-identifier | conformant-array-schema ) .
  index-type-specification =
        identifier ".." identifier
        ":" ordinal-type-identifier .
  bound-identifier = identifier .
  factor > bound-identifier .
```

NOTE. There is also a syntax rule for formal-parameter-section in 6.6.3.1. There is also a syntax rule for factor in 6.7.1.

If a conformant-array-schema contains a conformant-array-schema, then an abbreviated form of definition may be used. In the abbreviated form, a single semi-colon shall replace the sequence "]" "of" "array" "[" that occurs in the full form. The abbreviated form and the full form shall be equivalent.

Examples:

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

During the entire activation of the block, the first bound-identifier of an index-type-specification shall denote the smallest value specified by the corresponding index-type (see 6.6.3.8) possessed by each actual-parameter, and the second bound-identifier of the index-type-specification shall denote the largest value specified by that index-type.

The actual-parameters (see 6.7.3 and 6.8.2.3) corresponding to formal parameters that occur in a single conformant-array-parameter-specification shall all possess the same type. The type possessed by the actual-parameters shall be conformable (see 6.6.3.8) with the conformant-array-schema, and the formal parameters shall possess an array-type which shall be distinct from any other type, and which shall have a component-type that shall be that denoted by the type-identifier contained by the conformant-array-schema in the conformant-array-parameter-specification and which shall have the index-types of the type possessed by the actual-parameters that

correspond (see 6.6.3.8) to the index-type-specifications contained by the conformant-array-schema in the conformant-array-parameter-specification.

NOTE. The type of the formal parameter can not be a string-type (see 6.4.3.2) because it is not denoted by an array-type.

The actual-parameter shall be either a variable-access or an expression that is not a factor that is not a variable-access. If the actual-parameter is an expression, the value of the expression shall be attributed before activation of the block to an auxiliary variable which the program does not otherwise contain. The type possessed by this variable shall be the same as that possessed by the expression. This variable, or the actual-parameter if it is denoted by a variable-access, shall be accessed before the activation of the block, and this access shall establish a reference to the variable thereby accessed during the entire activation of the block; the corresponding formal parameter and its associated variable-identifier shall represent the referenced variable during the activation.

NOTE. In using an array variable A as an actual parameter corresponding to a formal parameter that occurs in a conformant-array-parameter-specification the use of an auxiliary variable is ensured by enclosing the variable-access A in parentheses.

An actual-parameter that is a variable-access shall not denote a component of a variable that possesses a type that is designated packed.

If the actual-parameter is an expression whose value is denoted by a variable-access that closest-contains an identifier which has a defining-occurrence in the identifier-list of a conformant-array-parameter-specification, then
(a) that identifier shall be contained by an indexed-variable contained by the expression, and
(b) the factor closest-containing the indexed-variable shall closest-contain at least as many index-expressions as the conformant-array-parameter-specification contains index-type-specifications.

NOTE. This ensures that the type of the expression and the anonymous variable will always be known and that, as a consequence, the activation record of a procedure can be of a fixed size.

6.6.3.8 Conformability.

NOTE. This clause does not apply to level 0.

Given a type denoted by an array-type closest-containing a single index-type, and a conformant-array-schema closest-containing a single index-type-specification, then the index-type and the

Second Draft Proposal

index-type-specification shall be designated as corresponding. Given two conformant-array-schemas closest-containing a single index-type-specification, then the two index-type-specifications shall be designated as corresponding. Let T1 be an array-type with a single index-type and let T2 be the type denoted by the ordinal-type-identifier of the index-type-specification of a conformant-array-schema closest-containing a single index-type-specification, then T1 shall be conformable with the conformant-array-schema if all the following four statements are true.

(a) The index-type of T1 is compatible with T2.
(b) The smallest and largest values specified by the index-type of T1 lie within the closed interval specified by T2.
(c) The component-type of T1 denotes the same type as that which is is denoted by the type-identifier of the conformant-array-schema, or is conformable to the conformant-array-schema in the conformant-array-schema.
(d) Either T1 is not designated packed and the conformant-array-schema is an unpacked-conformant-array-schema, or T1 is designated packed and the conformant-array-schema is a packed-conformant-array-schema .

NOTE. The abbreviated and full forms of a conformant-array-schema are equivalent (see 6.6.3.7). The abbreviated and full forms of an array-type are equivalent (see 6.4.3.2).

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

6.6.4 Required procedures and functions

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

NOTE. Required procedures and functions do not necessarily follow the rules given elsewhere for procedures and functions.

6.6.5 Required procedures

6,6.5.1 General. The required procedures shall be file handling procedures, dynamic allocation procedures and transfer procedures.

6.6.5.2 File handling procedures. Except for the application of rewrite or reset to the program parameters denoted by input or output, the effects of applying each of the file handling procedures rewrite, put, reset and get to a file-variable f shall be defined by pre-assertions and post-assertions about f, its components f.L, f.R, and f.M, and about the associated buffer-variable f^. The use of the variable f0 within an assertion shall be considered to represent the state or value, as appropriate, of f prior to the operation, and similarly for f0^ and f^, while f (within an assertion) shall denote the variable after the operation.

It shall be an error if the stated pre-assertion does not hold immediately prior to any use of the defined operation. It shall be an

Second Draft Proposal

error if any variable explicitly denoted in an assertion of equality is undefined. The post-assertion shall hold prior to the next subsequent access to the file, its components, or its associated buffer-variable. The post-assertions imply corresponding activities on the external entities, if any, to which the file-variables are bound. These activities, and the point at which they are actually performed, shall be implementation-defined.

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

put(f) pre-assertion: (f0.M = Generation) and
(f0.L is not undefined) and
(f0.R = S()) and
(f0^ is not undefined).
post-assertion: (f.M = Generation) and
(f.L = (f0.L~S(f0^))) and
(f.R = S()) and
(f^ is totally-undefined).

reset(f) pre-assertion: The components f0.L and f0.R are not undefined.
post-assertion: (f.L = S()) and
(f.R = (f0.L~f0.R~X)) and
(f.M = Inspection) and
(if f.R = S() then (f^ is totally-undefined)
else (f^ = f.R.first)),

where, if f is of the type denoted by the required structured-type-identifier text and if f0.L~f0.R is not empty and if (f0.L~f0.R).last is not designated an end-of-line, then X shall be a sequence having an end-of-line component as its only component; otherwise X = S().

get(f) pre-assertion: (f0.M = Inspection) and
(neither f0.L nor f0.R are undefined) and
(f0.R <> S()).

post-assertion: (f.M = Inspection) and
(f.L = (f0.L~S(f0.R.first))) and
(f.R = f0.R.rest) and
(if f.R = S() then (f^ is totally-undefined)
else (f^ = f.R.first)).

When the file-variable f possesses a type other than that denoted by text, the required procedures read and write shall be defined as follows.

Second Draft Proposal

read Read(f,v1,...,vn) where v1...vn denote variable-accesses shall be equivalent to

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

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

```
begin v := f^; get(f) end
```

NOTE. The variable-access is not a variable parameter. Consequently it may be a component of a packed structure and the value of the buffer-variable need only be assignment-compatible with it.

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

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

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

```
begin f^ := e; put(f) end
```

NOTES. 1. The required procedures read, write, readln, writeln, and page, as applied to textfiles, are described in 6.9.

2. Since the definitions of read and write include the use of get and put, the implementation-defined aspects of their post-assertions also apply.

6.6.5.3 Dynamic allocation procedures

new(p) shall create a new variable that is totally-undefined, shall create a new identifying-value of the pointer-type associated with p, that identifies the new variable, and shall attribute this identifying-value to the variable denoted by the variable-access p. The created variable shall possess the type that is the domain-type of the pointer-type possessed by p.

new(p,c1,...,cn) shall create a new variable that is totally-undefined, shall create a new identifying-value of the pointer-type associated with p, that identifies the new variable, and shall attribute this identifying-value to the variable denoted by the variable-access p. The created variable shall possess the record-type that is the domain-type of the pointer-type possesed by p and shall have nested variants that correspond to the case-constants c1,...,cn. The

case-constants shall be listed in order of increasing nesting of the variant-parts. Any variant not specified shall be at a deeper level of nesting than that specified by cn. It shall be an error if a variant of a variant-part within the new variable becomes active and a different variant of the variant-part is one of the specified variants.

dispose(q) shall remove the identifying-value denoted by the expression q from the pointer-type of q. It shall be an error if the identifying-value had been created using the form new(p,c1,...,cn).

dispose(q,k1,...,km) shall remove the identifying-value denoted by the expression q from the pointer-type of q. The case-constants k1,...,km shall be listed in order of increasing nesting of the variant-parts. It shall be an error if the variable had been created using the form new(p,c1,...,cn) and m is less than n. It shall be an error if the variants in the variable identified by q^ are different from those specified by the case-constants k1,...,km.

NOTE. The removal of an identifying-value from the pointer-type to which it belongs renders the identified variable inaccessible (see 6.5.4) and makes undefined all variables and functions that have that value attributed (see 6.8.2.2).

It shall be an error if q has a nil-value or is undefined.

It shall be an error if a variable created using the second form of new is accessed by the identified-variable of the variable-access of a factor, of an assignment-statement, or of an actual-parameter.

6.6.5.4 Transfer procedures. Let a be a variable possessing a type that can be denoted by
array [s1] of T,
let z be a variable possessing a type that can be denoted by
packed array [s2] of T,
and u and v be the smallest and largest values of the type s2, then the statement pack(a,i,z) shall be equivalent to

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

Second Draft Proposal

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

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

where j and k denote auxiliary variables which the program does not otherwise contain. The type possessed by j shall be s2, the type possessed by k shall be s1, and i shall be an expression whose value shall be assignment-compatible with s1.

6.6.6 Required functions
6.6.6.1 General. The required functions shall be arithmetic functions, transfer functions, ordinal functions and Boolean functions.

6.6.6.2 Arithmetic functions. For the following arithmetic functions, the expression x shall be either of real-type or integer-type. For the functions abs and sqr, the type of the result shall be the same as the type of the parameter, x. For the remaining arithmetic functions, the result shall always be of real-type.

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

6.6.6.3 Transfer functions
trunc(x) From the expression x that shall be of real-type, this function shall return a result of integer-type. The value of trunc(x) shall be such that if x is positive or zero then 0<=x-trunc(x)<1; otherwise -1<x-trunc(x)<=0. It shall be an error if such a value does not exist.
Examples:
trunc(3.5) yields 3
trunc(-3.5) yields -3
round(x) From the expression x that shall be of real-type, this function shall return a result of integer-type. If x is positive or zero, round(x) shall be equivalent to

trunc(x+0.5), otherwise round(x) shall be equivalent to trunc(x-0.5).
It shall be an error if such a value does not exist.
Examples:
round(3.5) yields 4
round(-3.5) yields -4

6.6.6.4 Ordinal functions

ord(x) From the expression x that shall be of an ordinal-type, this function shall return a result of integer-type that shall be the ordinal number (see 6.4.2.2 and 6.4.2.3) of the value of the expression x.

chr(x) From the expression x that shall be of integer-type, this function shall return a result of char-type which shall be the value whose ordinal number is equal to the value of the expression x if such a character value exists. It shall be an error if such a character value does not exist.

For any value, ch, of char-type, the following shall be true:
chr(ord(ch)) = ch

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

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

6.6.6.5 Boolean functions

odd(x) From the expression x that shall be of integer-type, this function shall be equivalent to the expression (abs(x) mod 2 = 1).

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

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

Second Draft Proposal

component (see 6.4.3.5), otherwise false.

6.7 Expressions

6.7.1 General. An expression shall denote a value unless a variable denoted by a variable-access contained by the expression is undefined at the time of its use, in which case that use shall be an error. The use of a variable-access as a factor shall denote the value, if any, attributed to the variable accessed thereby. Operator precedences shall be according to four classes of operators as follows. The operator not shall have the highest precedence, followed by the multiplying-operators, then the adding-operators and signs, and finally, with the lowest precedence, the relational-operators. Sequences of two or more operators of the same precedence shall be left associative.

```
unsigned-constant = unsigned-number | character-string |
                    constant-identifier | "nil" .
factor > variable-access | unsigned-constant |
         function-designator | set-constructor |
         "(" expression ")" | "not" factor .
```

NOTE. There is also a syntax rule for factor in 6.6.3.7

```
set-constructor = "[" [ member-designator
         { "," member-designator } ] "]" .
member-designator = expression [ ".." expression ] .
term = factor { multiplying-operator factor } .
simple-expression = [ sign ] term { adding-operator term } .
expression =
    simple-expression [ relational-operator simple-expression ] .
```

Any factor whose type is S, where S is a subrange of T, shall be treated as of type T. Similarly, any factor whose type is set of S shall be treated as of the unpacked canonical set-of-T type, and any factor whose type is packed set of S shall be treated as of the canonical packed set-of-T type.

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

A set-constructor shall denote a value of a set-type. The set-constructor [] shall denote that value in every set-type that contains no members. A set-constructor containing one or more member-designators shall denote either a value of the unpacked canonical set-of-T type or, if the context so requires, the packed canonical set-of-T type, where T is the type of every expression of each member-designator of the set-constructor. The type T shall be an ordinal-type. The value denoted by the set-constructor shall contain zero or more members each of which shall be denoted by at least one member-designator of the set-constructor.

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

NOTE. The member-designator x..y denotes no members if the value

Second Draft Proposal

of x is greater than the value of y.

Examples are as follows:

(a) Factors:

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

(b) Terms:

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

(c) Simple expressions:

```
p or q
x+y
-x
hue1 + hue2
i*J + 1
```

(d) Expressions:

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

6.7.2 Operators
6.7.2.1 General

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

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

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

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

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

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

Table 2. Dyadic arithmetic operations

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

Table 3. Monadic arithmetic operations

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

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

A term of the form x/y shall be an error if y is zero, otherwise the value of x/y shall be the result of dividing x by y.

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

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

Second Draft Proposal

NOTE. Only for i >= 0 does the relation
(i div J) * J + i mod J = i
hold.

The required constant-identifier maxint shall denote an implementation-defined value of integer-type. This value shall satisfy the following conditions:
(a) All integral values in the closed interval from -maxint to +maxint shall be values of the integer-type.
(b) Any monadic operation performed on an integer value in this interval shall be correctly performed according to the mathematical rules for integer arithmetic.
(c) Any dyadic integer operation on two integer values in this same interval shall be correctly performed according to the mathematical rules for integer arithmetic, provided that the result is also in this interval.
(d) Any relational operation on two integer values in this same interval shall be correctly performed according to the mathematical rules for integer arithmetic.

The results of the real arithmetic operators and functions shall be approximations to the corresponding mathematical results. The accuracy of this approximation shall be implementation-defined.

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

6.7.2.3 Boolean operators. Operands and results for Boolean operations shall be of Boolean-type. Boolean operators or, and and not shall denote respectively the logical operations of disjunction, conjunction and negation.

Boolean-expression = expression .

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

6.7.2.4 Set operators. The types of operands and results for set operations shall be as shown in table 4.

Table 4. Set operations

operator	operation	type of operands	type of result
+	set union	))a	))
-	set difference	)canonical	)same as the
*	set intersection	)set-of-T type)	)operands)

6.7.2.5 Relational operators. The types of operands and results for relational operations shall be as shown in table 5.

Table 5. Relational operations

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

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

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

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

Where u and v denote simple-expressions of a set-type, u <= v shall denote the inclusion of u in v and u >= v shall denote the inclusion of v in u.

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

When the relational operators = , <> , < , > , <= , >= are used to compare operands of compatible string-types (see 6.4.3.2), they denote lexicographic relations defined below. Lexicographic ordering imposes a total ordering on values of a string-type. If s1 and s2 are two values of compatible string-types then,

```
s1 = s2 iff for all i in [1..n]: s1[i] = s2[i]

s1 < s2 iff there exists a p in [1..n]:
       (for all i in [1..p-1]: s1[i] = s2[i]) and s1[p] < s2[p]
```

Second Draft Proposal

The operator in shall yield the value true if the value of the operand of ordinal-type is a member of the value of the set-type, otherwise it shall yield the value false.

6.7.3 Function designators. A function-designator shall yield the value of the function denoted by the function-identifier of the function-designator. The function-designator shall specify the activation of the function. If the function has any formal parameters the function-designator shall contain a list of actual-parameters that shall be bound to their corresponding formal parameters defined in the function-declaration. The correspondence shall be established by the positions of the parameters in the lists of actual and formal parameters respectively. The number of actual-parameters shall be equal to the number of formal parameters. The types of the actual-parameters shall correspond to the types of the formal parameters as specified by 6.6.3. The order of evaluation, accessing and binding of the actual-parameters shall be implementation-dependent.

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

Examples:

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

6.8 Statements

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

A label occurring in a statement S shall be designated as prefixing S, and shall be allowed to occur in a goto-statement G (see 6.8.2.4) if and only if any of the following three conditions is satisfied.

(a) S contains G.
(b) S is a statement of a statement-sequence containing G.
(c) S is a statement of the statement-sequence of the compound-statement of the statement-part of a block containing G.

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

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

6.8.2 Simple-statements

6.8.2.1 General. A simple-statement shall be a statement not containing a statement. An empty-statement shall contain no symbol and shall denote no action.

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

6.8.2.2 Assignment-statements. An assignment-statement shall attribute the value of the expression of the assignment-statement either to the variable denoted by the variable-access of the assignment-statement, or to the function-identifier of the assignment-statement; the value shall be assignment-compatible with the type possessed by the variable or function-identifier. The function-block associated (6.6.2) with the function-identifier of an assignment-statement shall contain the assignment-statement.

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

The decision as to the order of accessing the variable and evaluating the expression shall be implementation-dependent; the access shall establish a reference to the variable during the remaining execution of the assignment-statement.

The state of a variable or function when the variable or function does not have attributed to it a value specified by its type shall be designated undefined. If a variable possesses a structured-type, the state of the variable when every component of the variable is totally-undefined shall be designated totally-undefined. Totally-undefined shall be synonymous with undefined if the variable does not possess a structured-type.

Examples:
```
x := y+z
p := (1<=i) and (i<100)
```

Second Draft Proposal

```
i := sqr(k) - (i*j)
hue1 := [blue,succ(c)]
p1^.mother := true
```

6.8.2.3 Procedure-statements. A procedure-statement shall specify the activation of the block of the procedure-block associated with the procedure-identifier of the procedure-statement. If the procedure has any formal parameters the procedure-statement shall contain an actual-parameter-list, which is list of actual-parameters that shall be bound to their corresponding formal parameters defined in the procedure-declaration. The correspondence shall be established by the positions of the parameters in the lists of actual and formal parameters respectively. The number of actual-parameters shall be equal to the number of formal parameters. The types of the actual-parameters shall correspond to the types of the formal parameters as specified by 6.6.3. The order of evaluation, accessing and binding of the actual-parameters shall be implementation-dependent.

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

Examples:
```
printheading
transpose(a,n,m)
bisect(fct,-1.0,+1.0,x)
AddVectors(m[1],(m[2]),(m[k]))
```

6.8.2.4 Goto-statements. A goto-statement shall indicate that further processing is to continue at the program-point denoted by the label in the goto-statement and shall cause the termination of all activations except
(a) the activation containing the program-point and
(b) any activation containing the activation-point of an activation required by these exceptions not to be terminated.

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

6.8.3 Structured-statements
6.8.3.1 General.

```
structured-statement =
        compound-statement | conditional-statement|
        repetitive-statement | with-statement .
statement-sequence = statement { ";" statement } .
```

The execution of a statement-sequence specifies the execution of the statements of the statement-sequence in textual order, except as modified by execution of a goto-statement.

6.8.3.2 Compound-statements. A compound-statement shall specify execution of the statement-sequence of the compound-statement.

```
compound-statement = "begin" statement-sequence "end" .
```

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

6.8.3.3 Conditional-statements.

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

6.8.3.4 If-statements

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

If the Boolean-expression of the if-statement yields the value true, the statement of the if-statement shall be executed. If the Boolean-expression yields the value false, the statement of the if-statement shall not be executed and the statement of the else-part (if any) shall be executed.

An if-statement without an else-part shall not be followed by the token else.

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

Examples:

```
if x < 1.5 then z := x+y else z := 1.5

if p1 <> nil then p1 := p1^.father

if j = 0 then
  if i = 0 then writeln('indefinite')
  else writeln('infinite')
else writeln( i / j )
```

6.8.3.5 Case-statements. The values denoted by the case-constants of the case-constant-lists of the case-list-elements of a case-statement shall be distinct and of the same ordinal-type as the expression of the case-index of the case-statement. On execution of the case-statement the case-index shall be evaluated. That value shall then specify execution of the statement of the case-list-element closest-containing the case-constant denoting that value. One of the case-constants shall be equal to the value of the case-index upon entry to the case-statement.

It shall be an error if none of the case-constants is equal to the value of the case-index upon entry to the case-statement.

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

Second Draft Proposal

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

Example:

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

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

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

6.8.3.7 Repeat-statements

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

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

Example:

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

6.8.3.8 While-statements

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

The while-statement

```
while b do body
```

shall be equivalent to

Second Draft Proposal

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

Examples:

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

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

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

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

The control-variable shall be an entire-variable whose identifier is declared in the variable-declaration-part of the block closest-containing the for-statement. The control-variable shall possess an ordinal-type, and the initial-value and final-value shall be of a type compatible with this type. The statement of a for-statement shall not contain an assigning-reference (see 6.5.1) to the control-variable of the for-statement. The value of the final-value shall be assignment-compatible with the control-variable when the initial-value is assigned to the control-variable. After a for-statement is executed (other than being left by a goto-statement leading out of it) the control-variable shall be undefined. Apart from the restrictions imposed by these requirements, the for-statement

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

shall be equivalent to

Second Draft Proposal

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

and the for-statement

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

shall be equivalent to

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

where temp1 and temp2 denote auxiliary variables that the program does not otherwise contain, and that possess the type possessed by the variable v if that type is not a subrange-type; otherwise the host type of the type possessed by the variable v.

Examples:

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

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

for i:= 1 to 10 do
  for j := 1 to i-1 do
    m[i][j] := 0.0

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

6.8.3.10 With-statements

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

A with-statement shall specify the execution of the statement of the with-statement. The occurrence of a record-variable as the only record-variable in the record-variable-list of a with-statement shall constitute a defining-point of each of the field-identifiers associated with components of the record-type possessed by the record-variable as a field-designator-identifier for the region which is the statement of the with-statement; each applied occurrence of a field-designator-identifier shall denote that component of the record-variable which is associated with the field-identifier by the record-type. The record-variable shall be accessed before the statement of the with-statement is executed, and that access shall establish a reference to the variable during the entire execution of the statement of the with-statement.

The statement

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

shall be equivalent to

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

Second Draft Proposal

Example:

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

shall be equivalent to

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

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

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

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

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

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

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

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

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

```
begin v := f^; get(f) end
```

NOTE. The variable-access is not a variable parameter. Consequently it may be a component of a packed structure and the value of the buffer-variable need only be assignment-compatible with it.

(c) If v is a variable-access possessing the integer-type (or subrange thereof), read(f,v) shall cause the readings from f of a sequence of characters. Preceding spaces and end-of-lines shall be skipped. It shall be an error if the rest of the sequence does not form a signed-integer according to the syntax of 6.1.5. The value of the signed-integer thus read shall be assignment-compatible with the type possessed by v, and shall be attributed to v. Readings shall cease as soon as the buffer-variable f^ does not have attributed to it a character contained by the longest sequence available that forms a signed-integer.

(d) If v is a variable-access possessing the real-type, read(f,v) shall cause the readings from f of a sequence of characters. Preceding spaces and end-of-lines shall be skipped. It shall be an error if the rest of the sequence does not form a signed-number according to the syntax of 6.1.5. The value denoted by the number thus read shall be attributed to the variable v. Readings shall cease as soon as the buffer-variable f^ does not have attributed to it a character contained by the longest sequence available that forms a signed-number.

(e) When read is applied to f, it shall be an error if the buffer-variable f^ is undefined or the pre-assertions for get do not hold (see 6.4.3.5).

Second Draft Proposal

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

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

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

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

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

readln(f) shall be equivalent to

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

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

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

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

If the file-variable is omitted, the procedure shall be applied to the required textfile output. When write is applied to a textfile f, it shall be an error if f is undefined or f.M = Inspection (see 6.4.3.5). An application of write to a textfile f shall cause the buffer-variable f^ to become undefined.

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

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

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

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

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

where e is an expression whose value is to be written on the file f and may be of integer-type, real-type, char-type, Boolean-type or a string-type, and where TotalWidth and FracDigits are expressions of

integer-type whose values are the field-width parameters. The values of TotalWidth and FracDigits shall be greater than or equal to one; it shall be an error if either value is less than one.

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

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

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

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

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

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

and let IntDigits be the positive integer defined by:

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

then the representation shall consist of:

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

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

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

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

Second Draft Proposal

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

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

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

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

let ActWidth be the positive integer defined by:

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

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

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

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

```
the sign character,
   ( '-' if (e < 0) and (eWritten > 0), otherwise a space )
the leading digit-character of the decimal
   representation of eWritten,
the character '.' ,
the next DecPlaces digit-characters
   of the decimal representation of eWritten,
```

Second Draft Proposal

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

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

let eWritten be the non-negative number defined by:

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

let IntDigits be the positive integer defined by:

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

and let MinNumChars be the positive integer defined by:

```
MinNumChars := IntDigits + FracDigits + 1;
 if (e < 0.0) and (eWritten > 0)
    then MinNumChars := MinNumChars + 1;{'-' required}
```

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

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

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

Second Draft Proposal

6.9.4.6 Boolean-type. If e is of Boolean-type, a representation of the word true or the word false (as appropriate to the value of e) shall be written on the file f. This shall be equivalent to writing the appropriate character-strings 'True' or 'False' (see 6.9.4.7), where the case of each letter is implementation-defined, with a field-width parameter of TotalWidth.

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

if TotalWidth > n,
(TotalWidth - n) spaces,
the first through nth characters of the value of e in that order.

if 1 <= TotalWidth <= n,
the first through TotalWidththth characters in that order.

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

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

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

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

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

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

pre-assertion: (f0 is not undefined) and (f0.M = Generation).
post-assertion: (f.L = (f0.L~S(e))) and
(f^ is totally-undefined) and
(f.R = S()) and (f.M = Generation),
where S(e) is the sequence consisting solely of the end-of-line component defined in 6.4.3.5.

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

6.9.6 The procedure page. It shall be an error if the pre-assertion required for writeln(f) (see 6.9.5) does not hold prior to the activation of page(f). If the actual-parameter-list is omitted the procedure shall be applied to the required textfile output. Page(f) shall cause an implementation-defined effect on the textfile f, such that subsequent text written to f will be on a new page if the

textfile is printed on a suitable device, shall perform an implicit writeln(f) if f.L is not empty and if f.L.last is not the end-of-line component (see 6.4.3.5), and shall cause the buffer-variable f^ to become totally-undefined. The effect of inspecting a textfile to which the page procedure was applied during generation shall be implementation-dependent.

6.10 Programs.

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

The identifier of the program-heading shall be the program name which shall have no significance within the program. The identifiers contained by the program-parameters shall be distinct and shall be designated program parameters. Each program parameter shall be declared in the variable-declaration-part of the block of the program-block. The binding of the variables denoted by the program parameters to entities external to the program shall be implementation-dependent, except if the variable possesses a file-type in which case the binding shall be implementation-defined.

NOTE. The external representation of such external entities is not defined by this standard, nor is any property of a Pascal program dependent on such representation.

The occurrence of the identifier input or the identifier output as a program parameter shall constitute its defining-point for the region that is the program-block as a variable-identifier of the required type denoted by text. Such occurrence of the identifier input shall cause the post-assertions of reset to hold, and of output, the post-assertions of rewrite to hold, prior to the first access to the textfile or its associated buffer-variable. The effect of the application of the required procedure reset or the required procedure rewrite to either of these textfiles shall be implementation-defined.

Examples:

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

Second Draft Proposal

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

Second Draft Proposal

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

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

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

procedure q(procedure f; procedure g);
  begin
  f;
  g
  end { of q};

begin
globalone := 1;
globaltwo := 0;
p(q,dummy)
end.
```

6.11 Hardware representation. The representation for lexical tokens and separators given in 6.1 constitutes a reference representation for

program interchange. A processor shall accept all the reference symbols and all the alternative symbols except for any symbol whose representation contains a character not available in the character set of the processor. The reference symbols and the alternative symbols are given in table 6.

Table 6. Alternative symbols

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

NOTES. 1. The alternative comment delimiters are equivalent to the reference comment delimiters, thus a comment may begin with "{" and close with "*)", or begin with "(*" and close with "}".

2. For any other purpose than program interchange, this representation is not required, and so does not exclude the existence of other alternative symbols.

APPENDIX A. COLLECTED SYNTAX

```
actual-parameter =  expression | variable-access |
                      procedure-identifier |
                      function-identifier .

actual-parameter-list =
       "(" actual-parameter { "," actual-parameter } ")" .

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

apostrophe-image = "''" .

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

array-variable = variable-access .

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

base-type = ordinal-type .

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

Boolean-expression = expression .

bound-identifier = identifier .

buffer-variable = file-variable "^" .

case-constant = constant .

case-constant-list = case-constant { "," case-constant } .

case-index = expression .

case-list-element = case-constant-list ":" statement .

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

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

component-type = type-denoter .

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

compound-statement = "begin" statement-sequence "end" .

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

Second Draft Proposal

```
conformant-array-parameter-specification =
       "var" identifier-list ":" conformant-array-schema .

conformant-array-schema =
       (packed-conformant-array-schema |
       unpacked-conformant-array-schema) .

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

constant-definition = identifier "=" constant .

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

constant-identifier = identifier .

control-variable = entire-variable .

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

digit-sequence = digit {digit} .

directive = letter {letter | digit} .

domain-type = type-identifier .

else-part = "else" statement .

empty-statement = .

entire-variable = variable-identifier .

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

expression =
    simple-expression [ relational-operator simple-expression ] .

factor = variable-access | unsigned-constant | bound-identifier |
         function-designator | set-constructor |
         "(" expression ")" | "not" factor .

field-designator = record-variable "." field-specifier |
         field-designator-identifier .

field-identifier = identifier .

field-list =
    [ (fixed-part [ ";" variant-part ] | variant-part) [";"] ] .

field-specifier = field-identifier .
```

Second Draft Proposal

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

file-variable = variable-access .

final-value = expression .

fixed-part = record-section { ";" record-section } .

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

formal-parameter-list =
       "(" formal-parameter-section
       {";" formal-parameter-section} ")" .

formal-parameter-section =
       value-parameter-specification |
       variable-parameter-specification |
       procedural-parameter-specification |
       functional-parameter-specification |
       conformant-array-parameter-specification .

function-block = block .

function-declaration =
        function-heading ";" directive |
        function-identification ";" function-block |
        function-heading ";" function-block .

function-designator = function-identifier
                      [ actual-parameter-list ] .

function-heading =
        "function" identifier [formal-parameter-list]
        ":" result-type  .

function-identification =
        "function" function-identifier .

function-identifier = identifier .

functional-parameter-specification =
        function-heading .

goto-statement = "goto" label .

identified-variable = pointer-variable "^" .

identifier = letter {letter | digit} .

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

Second Draft Proposal

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

index-expression = expression .

index-type = ordinal-type .

index-type-specification =
               identifier ".." identifier
               ":" ordinal-type-identifier .

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

initial-value = expression .

label = digit-sequence .

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

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" .

member-designator = expression [ ".." expression ] .

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

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

new-pointer-type = "^" domain-type .

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

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

ordinal-type = new-ordinal-type |
                 integer-type | Boolean-type | char-type |
                 ordinal-type-identifier .

ordinal-type-identifier = identifier .

packed-conformant-array-schema =
                 "packed" "array" "[" index-type-specification "]"
                 "of" type-identifier .

pointer-type = new-pointer-type | pointer-type-identifier .

pointer-type-identifier = type-identifier .

pointer-variable = variable-access .
```

Second Draft Proposal

```
procedural-parameter-specification =
              procedure-heading .

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

procedure-block = block .

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

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

procedure-identification =
         "procedure" procedure-identifier .

procedure-identifier = identifier .

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

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

program-block = block .

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

program-parameters = identifier-list .

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

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

record-section = identifier-list ":" type-denoter .

record-type = "record" field-list "end" .

record-variable = variable-access .

record-variable-list =
       record-variable { "," record-variable } .

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

Second Draft Proposal

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

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

result-type = simple-type-identifier |
      pointer-type-identifier .

scale-factor = signed-integer .

set-constructor = "[" [ member-designator
                      { "," member-designator } ] "]" .

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

sign = "+" | "-" .

signed-integer = [sign] unsigned-integer .

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

signed-real = [sign] unsigned-real .

simple-expression = [ sign ] term { adding-operator term } .

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

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

simple-type-identifier = type-identifier .

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

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

statement-part = compound-statement .

statement-sequence = statement { ";" statement } .

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

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

Second Draft Proposal

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

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

structured-type-identifier = type-identifier .

subrange-type = constant ".." constant .

tag-field = identifier .

tag-type = ordinal-type-identifier .

term = factor { multiplying-operator factor } .

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

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

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

type-identifier = identifier .

unpacked-conformant-array-schema =
    "array" "[" index-type-specification
    { ";" index-type-specification } "]" "of"
    ( type-identifier | conformant-array-schema ) .

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

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

unsigned-integer = digit-sequence .

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

unsigned-real =
    unsigned-integer "." digit-sequence ["e" scale-factor] |
    unsigned-integer "e" scale-factor .

value-parameter-specification =
    identifier-list ":" type-identifier .

variable-access = entire-variable | component-variable |
            identified-variable | buffer-variable .

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

Second Draft Proposal

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

variable-identifier = identifier .

variable-parameter-specification =
             "var" identifier-list ":" type-identifier .

variant = case-constant-list ":"  "(" field-list ")" .

variant-part = "case" variant-selector "of"
                 variant { ";" variant } .

variant-selector = [tag-field ":"] tag-type .

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

with-statement =
       "with" record-variable-list "do"
       statement .

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" .

write-parameter = expression [":" expression [":" expression ] ] .

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

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

APPENDIX B. INDEX

Second Draft Proposal

Second Draft Proposal

Second Draft Proposal

Second Draft Proposal

Second Draft Proposal

APPENDIX C. REQUIRED IDENTIFIERS

IDENTIFIER	REFERENCE CLAUSE(S)
abs	6.6.6.2
arctan	6.6.6.2
Boolean	6.4.2.2
char	6.4.2.2
chr	6.6.6.4
cos	6.6.6.2
dispose	6.6.5.3
eof	6.6.6.5
eoln	6.6.6.5
exp	6.6.6.2
false	6.4.2.2
get	6.6.5.2
input	6.10
integer	6.4.2.2
ln	6.6.6.2
maxint	6.7.2.2
new	6.6.5.3
odd	6.6.6.5
ord	6.6.6.4
output	6.10
pack	6.6.5.4
page	6.9.6
pred	6.6.6.4
put	6.6.5.2
read	6.6.5.2, 6.9.2
readln	6.9.3
real	6.4.2.2
reset	6.6.5.2
rewrite	6.6.5.2
round	6.6.6.3
sin	6.6,6.2
sqr	6.6.6.2
sqrt	6.6.6.2
succ	6.6.6.4
text	6.4.3.5
true	6.4.2.2
trunc	6.6.6.3
unpack	6.6.5.4
write	6.6.5.2, 6.9.4.1, 6.9.4.2
writeln	6.9.5

IMPLEMENTATION NOTES ONE PURPOSE COUPON

0. **DATE**

1. **IMPLEMENTOR/MAINTAINER/DISTRIBUTOR** *(* Give a person, address and phone number. *)*

2. **MACHINE/SYSTEM CONFIGURATION** *(* Any known limits on the configuration or support software required, e.g. operating system. *)*

3. **DISTRIBUTION** *(* Who to ask, how it comes, in what options, and at what price. *)*

4. **DOCUMENTATION** *(* What is available and where. *)*

5. **MAINTENANCE** *(* Is it unmaintained, fully maintained, etc? *)*

6. **STANDARD** *(* How does it measure up to standard Pascal? Is it a subset? Extended? How. *)*

7. **MEASUREMENTS** *(* Of its speed or space. *)*

8. **RELIABILITY** *(* Any information about field use or sites installed. *)*

9. **DEVELOPMENT METHOD** *(* How was it developed and what was it written in? *)*

10. **LIBRARY SUPPORT** *(* Any other support for compiler in the form of linkages to other languages, source libraries, etc. *)*

(FOLD HERE)

PLACE
POSTAGE
HERE

Bob Dietrich
M.S. 92-134
Tektronix, Inc.
P.O. Box 500
Beaverton, Oregon 97077
U.S.A.

(FOLD HERE)

NOTE: Pascal News publishes all the checklists it gets. Implementors should send us their checklists for their products so the thousands of committed Pascalers can judge them for their merit. Otherwise we must rely on rumors.

Please feel free to use additional sheets of paper.

IMPLEMENTATION NOTES ONE PURPOSE COUPON

POLICY: PASCAL USERS GROUP (15-Sep-80)

Purpose: The Pascal User's Group (PUG) promotes the use of the programming language Pascal as well as the ideas behind Pascal through the vehicle of Pascal News. PUG is intentionally designed to be non political, and as such, it is not an "entity" which takes stands on issues or support causes or other efforts however well-intentioned. Informality is our guiding principle; there are no officers or meetings of PUG.

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

Membership: Anyone can join PUG, particularly the Pascal user, teacher, maintainer, implementor, distributor, or just plain fan. Memberships from libraries are also encouraged. See the ALL-PURPOSE COUPON for details.

Facts about Pascal, THE PROGRAMMING LANGUAGE:

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

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

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

Pascal implementations exist for more than 105 different computer systems, and this number increases every month. The "Implementation Notes" section of Pascal News describes how to obtain them.

The standard reference and tutorial manual for Pascal is:

> Pascal - User Manual and Report (Second, study edition)
> by Kathleen Jensen and Niklaus Wirth.
> Springer-Verlag Publishers: New York, Heidelberg, Berlin
> 1978 (corrected printing), 167 pages, paperback, $7.90.

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

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

Remember, **Pascal User's Group** is each individual member's group. We currently have more than 3500 active members in more than 41 countries. this year Pascal News is averaging more than 100 pages per issue.

Policy