INTERNATIONAL STANDARD

ISO/IEC 7185

Second edition
1990-10-15

Information technology — Programming languages — Pascal

(Revision of ISO 7185 : 1983)

Technologies de l'information — Langages de programmation — Pascal

(Révision de l'ISO 7185 : 1983)

Reference number
ISO/IEC 7185 : 1990 (E)

Contents

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work.

In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75 % of the national bodies casting a vote.

International Standard ISO/IEC 7185 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology.*

This second edition cancels and replaces the first edition (ISO 7185 : 1983).

Annexes A, B, C, D, E and F are for information.

Introduction

This International Standard provides an unambiguous and machine independent definition of the programming language Pascal. Its purpose is to facilitate portability of Pascal programs for use on a wide variety of data processing systems.

Language history

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

a) to make available a language suitable for teaching programming as a systematic discipline based on certain fundamental concepts clearly and naturally reflected by the language;

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

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

Project history

In 1977, a working group was formed within the British Standards Institution (BSI) to produce a standard for the programming language Pascal. This group produced several working drafts, the first draft for public comment being widely published early in 1979. In 1978, BSI's proposal that Pascal be added to ISO's program of work was accepted, and the ISO Pascal Working Group (then designated ISO/TC97/SC5/WG4) was formed in 1979. The Pascal standard was to be published by BSI on behalf of ISO, and this British Standard referenced by the International Standard.

In the USA, in the fall of 1978, application was made to the IEEE Standards Board by the IEEE Computer Society to authorize project 770 (Pascal). After approval, the first meeting was held in January 1979.

In December of 1978, X3J9 convened as a result of a SPARC (Standards Planning and Requirements Committee) resolution to form a US TAG (Technical Advisory Group) for the ISO Pascal standardization effort initiated by the UK. These efforts were performed under X3 project 317.

In agreement with IEEE representatives, in February of 1979, an X3 resolution combined the X3J9 and P770 committees into a single committee called the Joint X3J9/IEEE-P770 Pascal Standards Committee. (Throughout, the term JPC refers to this committee.) The first meeting as JPC was held in April 1979.

The resolution to form JPC clarified the dual function of the single joint committee to produce a dpANS and a proposed IEEE Pascal standard, identical in content.

ANSI/IEEE770X3.97-1983, American National Standard Pascal Computer Programming Language, was approved by the IEEE Standards Board on September 17, 1981, and by the American National Standards

Institute on December 16, 1982. British Standard BS6192, Specification for Computer programming language Pascal, was published in 1982, and International Standard 7185 (incorporating BS6192 by reference) was approved by ISO on December 1, 1983. Differences between the ANSI and ISO standards are detailed in the Foreword of ANSI/IEEE770X3.97-1983.

In 1985, the ISO Pascal Working Group (then designated ISO/TC97/SC22/WG2, now ISO/IEC JTC1/ SC22/WG2) was reconvened after a long break. An Interpretations Subgroup was formed, to interpret doubtful or ambiguous portions of the Pascal standards. As a result of the work of this subgroup, and also of the work on the Extended Pascal standard being produced by WG2 and JPC, BS6192/ISO7185 was revised and corrected during 1988/89; it is expected that ANSI/IEEE770X3.97-1983 will be replaced by the revised ISO 7185.

The major revisions to BS6192:1982 to produce the new ISO 7185 are:

a) resolution of the differences with ANSI/IEEE770X3.97-1983;

b) relaxation of the syntax of real numbers, to allow "digit sequences" rather than "unsigned integers" for the various components;

c) in the handling of "end-of-line characters" in text files;

d) in the handling of run-time errors.

This page intentionally left blank

Information technology — Programming languages — Pascal

1 Scope

1.1

This International 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 International 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, nor the actions to be taken when the corresponding limits are exceeded;

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 Normative reference

The following standard contains provisions which, through reference in this text, constitute provisions of this International Standard. At the time of publication, the edition indicated was valid. All standards are subject to revision, and parties to agreements based on this International Standard are encouraged to investigate the possibility of applying the most recent edition of the standard listed below. Members of IEC and ISO maintain registers of currently valid International Standards.

ISO 646:1983, *Information processing—ISO 7-bit coded character set for information interchange.*

3 Definitions

For the purposes of this International Standard, the following definitions apply.

NOTE — To draw attention to language concepts, some terms are printed in italics on their first mention or at their defining occurrence(s) in this International Standard.

3.1 Error

A violation by a program of the requirements of this International Standard that a processor is permitted to leave undetected.

NOTES

1 If it is possible to construct a program in which the violation or non-violation of this International Standard requires knowledge of the data read by the program or the implementation definition of implementation-defined features, then violation of that requirement is classified as an error. Processors may report on such violations of the requirement without such knowledge, but there always remain some cases that require execution, simulated execution, or proof procedures with the required knowledge. Requirements that can be verified without such knowledge are not classified as errors.

2 Processors should attempt the detection of as many errors as possible, and to as complete a degree as possible. Permission to omit detection is provided for implementations in which the detection would be an excessive burden.

3.2 Extension

A modification to clause **6** of the requirements of this International Standard that does not invalidate any program complying with this International Standard, as defined by **5.2**, except by prohibiting the use of one or more particular spellings of identifiers (see **6.1.2** and **6.1.3**).

3.3 Implementation-defined

Possibly differing between processors, but defined for any particular processor.

3.4 Implementation-dependent

Possibly differing between processors and not necessarily defined for any particular processor.

3.5 Processor

A system or mechanism that accepts a program as input, prepares it for execution, and executes the process so defined with data to produce results.

NOTE — A processor may consist of an interpreter, a compiler and run-time system, or another mechanism, together with an associated host computing machine and operating system, or another mechanism for achieving the same effect. A compiler in itself, for example, does not constitute a processor.

4 Definitional conventions

The metalanguage used in this International Standard to specify the syntax of the constructs is based on Backus-Naur Form. The notation has been modified from the original to permit greater convenience of description and to allow for iterative productions to replace recursive ones. Table 1 lists the meanings of the various metasymbols. 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 a required identifier shall denote the corresponding required 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 International Standard.

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

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

Table 1 — Metalanguage symbols

Metasymbol	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 nonterminal symbol

The characters required to form Pascal programs shall be 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 nonterminal 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 y from which an x is directly or indirectly derived;

—the y *closest-containing* an x: that y containing an x and not containing 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.

In addition to the normal English rules for hyphenation, hyphenation is used in this International Standard to form compound words that represent meta-identifiers, semantic terms, or both. All meta-identifiers that contain more than one word are written as a unit with hyphens joining the parts. Semantic terms ending in "type" and "variable" are also written as one hyphenated unit. Semantic terms representing compound ideas are likewise written as hyphenated units, e.g., digit-value, activation-point, assignment-compatible, and identifying-value.

NOTES are included in this International Standard only for purposes of clarification, and aid in the use of the standard. NOTES are informative only and are not a part of the International Standard.

Examples in this International Standard are equivalent to NOTES.

5 Compliance

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 International 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) be able to determine whether or not the program violates any requirements of this International Standard, where such a violation is not designated an error, report the result of this determination to the user of the processor before the execution of the program-block, if any, and shall prevent execution of the program-block, if any;

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, and a note referencing each such statement shall appear in a separate section of the accompanying document;

2) the processor shall report the error or the possibility of the error during preparation of the program for execution and in the event of such a report shall be able to continue further processing and shall be able to refuse execution of the program-block;

3) the processor shall report the error during execution of the program;

and if an error is reported during execution of the program, the processor shall terminate execution; if an error occurs within a statement, the execution of that statement shall not be completed;

NOTE — 1 This means that processing will continue up to or beyond execution of the program at the option of the user.

g) be accompanied by a document that separately describes any features accepted by the processor that are prohibited or not specified in clause **6**: such extensions shall be described as being 'extensions to Pascal as specified by ISO/IEC 7185';

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.

NOTE — 2 The phrase 'be able to' is used in **5.1** to permit the implementation of a switch with which the user may control the reporting.

A processor that purports to comply, wholly or partially, with the requirements of this International Standard shall do so only in the following terms. A *compliance statement* shall be produced by the processor as a consequence of using the processor or shall be included in accompanying documentation. If the processor complies in all respects with the requirements of this International Standard, the compliance statement shall be

<*This processor*> complies with the requirements of level <*number*> of ISO/IEC 7185.

If the processor complies with some but not all of the requirements of this International Standard then it shall not use the above statement, but shall instead use the following compliance statement

<*This processor*> complies with the requirements of level <*number*> of ISO/IEC 7185, with the following exceptions: <*followed by a reference to, or a complete list of, the requirements of the International Standard with which the processor does not comply*>.

In both cases the text <*This processor*> shall be replaced by an unambiguous name identifying the processor, and the text <*number*> shall be replaced by the appropriate level number.

NOTE — 3 Processors that do not comply fully with the requirements of the International Standard are not required to give full details of their failures to comply in the compliance statement; a brief reference to accompanying documentation that contains a complete list in sufficient detail to identify the defects is sufficient.

5.2 Programs

A program conforming with the requirements of this International Standard shall

a) if it conforms 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 conforms at level 1, use only those features of the language specified in clause **6**; and

c) not rely on any particular interpretation of implementation-dependent features.

NOTES

1 A program that complies with the requirements of this International Standard may rely on particular implementation-defined values or features.

2 The requirements for conforming programs and compliant processors do not require that the results produced by a conforming program are always the same when processed by a compliant processor. They may be the same, or they may differ, depending on the program. A simple program to illustrate this is

```
program x(output); begin writeln(maxint) end.
```

6 Requirements

6.1 Lexical tokens

NOTE — The syntax given in this subclause 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 International Standard.

6.1.1 General

The lexical tokens used to construct Pascal programs are 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 are 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 can be of any length. The *spelling* of an identifier shall be composed from all its constituent characters taken in textual order, without regard for the case of letters. No identifier shall have the same spelling as any word-symbol. Identifiers that are specified to be *required* shall have special significance (see **6.2.2.10** and **6.10**).

identifier = letter { letter | digit } .

Examples:

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

6.1.4 Directives

A directive shall only occur in a procedure-declaration or a function-declaration. The only directive shall be the required directive **forward** (see **6.6.1** and **6.6.2**). No directive shall have the same spelling as any word-symbol.

directive = letter { letter | digit } .

NOTE — Many processors provide, as an extension, the directive **external**, which 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**).

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

signed-real = [sign] unsigned-real .

signed-integer = [sign] unsigned-integer .

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

sign = ‘+’ | ‘−’ .

unsigned-real = digit-sequence ‘.’ fractional-part [‘e’ scale-factor]
| digit-sequence ‘e’ scale-factor .

unsigned-integer = digit-sequence .

fractional-part = digit-sequence .

scale-factor = [sign] digit-sequence .

digit-sequence = digit { digit } .

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 and shall be in the closed interval 0 to 9999. The *spelling* of a label shall be its apparent integral value.

label = digit-sequence .

6.1.7 Character-strings

A character-string containing a single string-element shall denote a value of the required 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. All character-strings with a given number of components shall possess the same string-type.

There shall be an implementation-defined one-to-one correspondence between the set of alternatives from which string-elements are drawn and a subset of the values of the required char-type. The occurrence of a string-element in a character-string shall denote the occurrence of the corresponding 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 .

NOTE — Conventionally, the apostrophe-image is regarded as a substitute for the apostrophe character, which cannot be a string-character.

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

6.1.8 Token separators

Where a *commentary* shall be any sequence of characters and separations of lines, containing neither } nor *), the construct

(‘{’ | ‘(*’) commentary (‘*)’ | ‘}’)

shall be a *comment* if neither the { nor the (* occurs within a character-string or within a commentary.

NOTES

1 A comment may thus commence with `{` and end with `*)`, or commence with `(*` and end with `}`.

2 The sequence `(*)` cannot occur in a commentary even though the sequence `{)` can.

Comments, spaces (except in character-strings), and the separations of consecutive lines shall be considered to be token separators. Zero or more token separators can occur between any two consecutive tokens, before the first token of a program text, or after the last token of the 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.

6.1.9 Lexical alternatives

The representation for lexical tokens and separators given in **6.1.1** to **6.1.8**, except for the character sequences (* and *), shall constitute a *reference representation* for these tokens and separators.

To facilitate the use of Pascal on processors that do not support the reference representation, the following alternatives have been defined. All processors that have the required characters in their character set shall provide both the reference representations and the alternative representations, and the corresponding tokens or separators shall not be distinguished. Provision of the reference representations, and of the alterative token @, shall be implementation-defined.

The alternative representations for the tokens shall be

Reference token	**Alternative token**
↑	@
[	(.
]	.)

NOTE — 1 The character ↑ that appears in some national variants of ISO 646 is regarded as identical to the character ^. In this International Standard, the character ↑ has been used because of its greater visibility.

The comment-delimiting characters { and } shall be the reference representations, and (* and *) respectively shall be alternative representations (see **6.1.8**).

NOTE — 2 See also **1.2 f)**.

6.2 Blocks, scopes, and activations

6.2.1 Blocks

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 for the region that is the block. Each applied occurrence of that label (see **6.2.2.8**) shall be a label. Within an activation of the block, all applied occurrences of that label shall denote the corresponding program-point in the algorithm of the activation at that statement (see **6.2.3.2 b)**).

```
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 .

6.2.2 Scopes

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**, **6.10**).

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 has a defining-point for region A and another identifier or label having the same spelling has a 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 region that is the field-specifier of a field-designator shall be excluded from the enclosing scopes.

6.2.2.7

When an identifier or label has a defining-point for a region, another identifier or label with the same spelling shall not have a defining-point for that region.

6.2.2.8

Within the scope of a defining-point of an identifier or label, each occurrence of an identifier or label having the same spelling as the identifier or label of the defining-point shall be designated an *applied occurrence* of the identifier or label of the defining-point, except for an occurrence that constituted the defining-point; such an occurrence shall be designated a *defining occurrence*. No occurrence outside that scope shall be an applied occurrence.

NOTE — Within the scope of a defining-point of an identifier or label, there are no applied occurrences of an identifier or label that cannot be distinguished from it and have a defining-point for a region enclosing that scope.

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 an identifier can have an applied occurrence in the type-identifier of the domain-type of any new-pointer-types contained by the type-definition-part containing the defining-point of the type-identifier.

6.2.2.10

Required identifiers that denote required values, types, procedures, and functions shall be used as if their defining-points have a region enclosing the program (see **6.1.3**, **6.3**, **6.4.1**, and **6.6.4.1**).

NOTE — The required identifiers **input** and **output** are not included, since these denote variables.

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.

NOTES

1 Within syntax definitions, an applied occurrence of an identifier is qualified (e.g., type-identifier), whereas a use that constitutes a defining-point is not qualified.

2 It is intended that such qualification indicates the nature of the entity denoted by the applied occurrence: e.g., a constant-identifier denotes a constant.

6.2.3 Activations

6.2.3.1

A procedure-identifier or function-identifier having a defining-point for a region that 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 defining-point of a label in the label-declaration-part of the block, a corresponding program-point (see **6.2.1**);

c) for each variable-identifier having a defining-point for the region that 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 procedure-block corresponding to the procedure-identifier, and the formal-parameters of that procedure-block;

e) for each function-identifier local to the block, a function with the function-block corresponding to, and the result type associated with, the function-identifier, and the formal-parameters of that function-block;

f) if the block is a function-block, a result possessing the associated result type.

NOTE — Each activation contains its own algorithm, program-points, variables, procedures, and functions, distinct from every other activation.

6.2.3.3

The activation of a procedure or function shall be an activation of the block of the procedure-block of the procedure or function-block of the function, respectively, and shall be designated as *within*

a) the activation containing the procedure or function; and

b) all activations that that containing activation is within.

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; except that the function-identifier of an assignment-statement shall, within an activation of the function denoted by that function-identifier, denote the result of that activation.

6.2.3.4

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

6.2.3.5

All variables contained by an activation, except for those listed as program-parameters, and any result of an activation, shall be totally-undefined at the commencement of that activation. 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 in a constant-definition 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. The required constant-identifiers shall be as specified in **6.4.2.2** and **6.7.2.2**.

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 the type-denoter of the type-definition. Except for applied occurrences in 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-types, structured-types or pointer-types. The required type-identifiers and corresponding required types shall be as specified in **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. A value of an ordinal-type shall have an integer ordinal number; the ordering relationship between any two such values of one type shall be the same as that between their ordinal numbers. An ordinal-type-identifier shall denote an ordinal-type. A real-type-identifier shall denote the real-type.

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

ordinal-type = new-ordinal-type | ordinal-type-identifier .

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

ordinal-type-identifier = type-identifier .

real-type-identifier = type-identifier .

6.4.2.2 Required simple-types

The following types shall exist

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

b) *real-type*. The required type-identifier **real** shall denote the real-type. The real-type shall be a simple-type. The values shall be an implementation-defined subset of the real numbers, denoted as specified in **6.1.5** by signed-real.

c) *Boolean-type*. The required type-identifier **Boolean** shall denote the Boolean-type. The Boolean-type shall be an ordinal-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.

d) *char-type*. The required type-identifier **char** shall denote the char-type. The char-type shall be an ordinal-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 following relations shall hold.

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

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

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

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

6.4.2.3 Enumerated-types

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

identifier-list = identifier { ',' identifier } .

The occurrence of an identifier in the identifier-list of an enumerated-type shall constitute its defining-point for the region that is the block closest-containing the enumerated-type. Each applied occurrence of the identifier shall be a constant-identifier. Within an activation of the block, all applied occurrences of that identifier shall possess the type denoted by the enumerated-type and shall denote the type's value whose ordinal number is the number of occurrences of identifiers preceding that identifier in the identifier-list.

NOTE — Enumerated type constants are ordered by the sequence in which they are defined, and they have consecutive ordinal numbers starting at zero.

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

A subrange-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 second 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 .

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

unpacked-structured-type = array-type | record-type | set-type | file-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 economized, 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; i.e., if a component is itself structured, the component's representation in data-storage shall be packed only if the type of the component is designated packed.

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

6.4.3.2 Array-types

An array-type shall be structured as a mapping from each value specified by its index-type to 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 .

Example 1:
```
    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 index-type in the sequence 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 — 1 Each of the following two examples thus contains different ways of expressing its array-type.

Example 2:
```
    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 3:

```
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, respectively; 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 — 2 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 a largest value of greater than 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 string-elements of the character-string, taken in textual order, to the components of the values of the string-type in order of increasing index.

NOTE — 3 The values of a string-type possess additional properties which allow writing them to textfiles (see **6.9.3.6**) 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 .

field-identifier = identifier .

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 containing 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 that 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 that shall have the values and structure defined by the variant-part.

Let V_i 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

$$(V_1, V_2, \ldots, V_m).$$

NOTE — 1 If the type of the *i*-th component has F_i values, then the cardinality of the set of values of the field-list is $(F_1 * F_2 * \ldots * F_m)$.

A tag-type shall be 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 that is constructed to possess exactly one value for each variant of the variant-part, and no others, and each such 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 that 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 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

$$(k, X_k)$$

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

NOTES

2 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 T_i values, then the cardinality of the set of values of the variant-part is $(T_1 + T_2 + \ldots + T_n)$. There is no component of a value of a variant-part corresponding to any non-active variant of the variant-part.

3 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 power set of the base-type of the set-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 — 1 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 — 2 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 each ordinal-type T that is not a subrange-type, there shall exist both an unpacked set-type designated the *unpacked-canonical-set-of-T-type* and a packed set-type designated the *packed-canonical-set-of-T-type*. If S is any subrange-type and T is its host-type, then the set of values determined by the type set of S shall be included in the sets of values determined by the unpacked-canonical-set-of-T-type and by the packed-canonical-set-of-T-type (see **6.7.1**).

6.4.3.5 File-types

NOTE — 1 A file-type describes sequences of values of the specified component-type, together with a current position in each sequence and a mode that 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 in items a) to e).

NOTE — 2 The notation x~y represents the concatenation of sequences x and y. The explicit representation of sequences (e.g., S(c)), of concatenation of sequences; of the first, last, and rest selectors; and of sequence equality is not part of the Pascal language. These notations are used to define file values, below, and the required file operations in **6.6.5.2** and **6.6.6.5**.

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

b) Let c be a value of the specified component-type and let x be a value of the sequence-type S; then S(c) shall be a sequence of type S, consisting of the single component-value c, and both S(c)~x and x~S(c) shall be sequences, 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 or y is the empty sequence, then x = y shall be true if and only if both x and y are 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 — 3 The explicit denotation of the values Inspection and Generation 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 and 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 — 4 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 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 cs ~S(end-of-line), where cs is a sequence of components possessing the char-type, and *end-of-line* shall represent a special component-value. Any assertion in clause **6** that the end-of-line value is attributed to a variable other than a component of a sequence shall be construed as an assertion that the variable has attributed to it the char-type value space. If l is a line, then no component of l other than l.last shall be an end-of-line. There shall be an implementation-defined subset of the set of char-type values, designated *characters prohibited from textfiles*; the effect of causing a character in that subset to be attributed to a component of either t.L or t.R for any textfile t shall be implementation-dependent.

A *line-sequence*, ls, shall be either the empty sequence or the sequence l ~ ls' where l is a line and ls' is a line-sequence.

Every value t of the type denoted by **text** shall satisfy 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 ls ~cs, where ls is a line-sequence and cs is a sequence of components possessing the char-type.

NOTE — 5 In rule b), cs may be considered, especially if it is non-empty, to be a partial line that is being generated. Such a partial line cannot occur during inspection of a file. Also, cs 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 type-identifier **text** shall be designated a *textfile*.

NOTE — 6 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 new-pointer-type. The set of identifying-values shall be dynamic, in that the variables and the values identifying them shall be permitted to 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 — 1 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 — 2 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 following four statements 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 following five statements is true:

a) T1 and T2 are the same type, and that type is permissible as the component-type of a file-type (see **6.4.3.5**).

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.

At any place where the rule of assignment-compatibility is used to require a value of integer-type to be assignment-compatible with a real-type, an implicit integer-to-real conversion shall be performed.

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;
 indextype = 1..limit;
 vector = array [indextype] of real;

 person = ↑ persondetails;

 persondetails = record
                  name, firstname : charsequence;
                  age : natural;
                  married : Boolean;
                  father, child, sibling : person;
```

```
                    case s : sex of
                     male :
                      (enlisted, bearded : Boolean);
                     female :
                      (mother, programmer : Boolean)
                  end;
 FileOfInteger = file of integer;
```

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

6.5 Declarations and denotations of variables

6.5.1 Variable-declarations

A variable shall be an entity to which a value can 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 a declared variable, a component of a variable, a variable that 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 .
```

Example of a variable-declaration-part:

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

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

NOTE — Variables occurring in examples in the remainder of this International Standard should be assumed to have been declared as specified in the above example.

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 or an access to a component of a variable shall constitute a reference or an 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**).

Example 1:
```
a[12]
a[i + j]
m[k]
```

If the array-variable is itself an indexed-variable, an abbreviation shall be permitted. 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.

The order of both the evaluation of the index-expressions of, and the access to the array-variable of, an indexed-variable shall be implementation-dependent.

Example 2:
```
m[k][1]
m[k, 1]
```

NOTE — These 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 associated (see **6.4.3.3**) with the field-identifier of the field-specifier of the field-designator 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 .

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 associated (see **6.4.3.3**) 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 non-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**) 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 that 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 the set of values of the 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↑.sibling↑.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 an access to a buffer-variable shall constitute a reference or an access, respectively, to the associated file-variable.

6.6 Procedure and function declarations

6.6.1 Procedure-declarations

```
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 procedure-declaration.

Each identifier having a defining-point as a procedure-identifier in a procedure-heading of a procedure-declaration in which the directive **forward** occurs shall have exactly one of its applied occurrences in a procedure-identification of a procedure-declaration, and this applied occurrence shall be closest-contained by the procedure-and-function-declaration-part closest-containing the procedure-heading.

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

There shall be at most one procedure-block associated with a procedure-identifier.

Examples of procedure-and-function-declaration-parts:

Example 1:

NOTE — This example is not for level 0.

```
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 };
```

Example 2:

```
procedure readinteger (var f : text; var x : integer);
  var
     i : natural;
  begin
  while f↑ = ' ' do get(f);
  {The buffer-variable 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 buffer-variable contains a non-digit}
  x := i
  {Of course if there are no digits, x is zero}
 end;

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 subinterval.  Return the midpoint as
   the zero.}
  result := midpoint
end;

procedure PrepareForAppending (var f : FileOfInteger);
{ This procedure takes a file in any state suitable for reset and
  places it 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

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 such that the function-identifier of the assignment-statement is associated with the block (see **6.8.2.2**).

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

Each identifier having a defining-point as a function-identifier in the function-heading of a function-declaration in which the directive **forward** occurs shall have exactly one of its applied occurrences in a function-identification of a function-declaration, and this applied occurrence shall be closest-contained by the procedure-and-function-declaration-part closest-containing the function-heading.

The occurrence of a function-block in a function-declaration shall associate the function-block with the identifier in the function-heading, or with the function-identifier in the function-identification, of the function-declaration; the block of the function-block shall be associated with the result type that is associated with the identifier or function-identifier.

There shall be at most one function-block associated with a function-identifier.

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, estimate : real;
begin
 estimate := x;
 repeat
   old := estimate;
   estimate := (old + x / old) * 0.5;
```

```
until abs(estimate - old) < eps * estimate;
{eps being a global constant}
Sqrt := estimate
end { of Sqrt };

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;


{The following two functions analyze a parenthesized expression and
 convert it to an internal form. They are declared forward
 since they are mutually recursive, i.e., they call each other.
 These function-declarations use the following identifiers that are not
 defined in this International Standard: formula, IsOpenParenthesis, IsOperator,
 MakeFormula, nextsym, operation, ReadElement, ReadOperator, and
 SkipSymbol. }

function ReadExpression : formula; forward;

function ReadOperand : formula; forward;

function ReadExpression; {See forward declaration of heading.}
var
  this : formula;
  op : operation;
begin
 this := ReadOperand;
 while IsOperator(nextsym) do
  begin
   op := ReadOperator;
   this := MakeFormula(this, op, ReadOperand);
  end;
 ReadExpression := this
end;
```

```
function ReadOperand; {See forward declaration of heading.}
begin
 if IsOpenParenthesis(nextsym) then
  begin
  SkipSymbol;
  ReadOperand := ReadExpression;
  {nextsym should be a close-parenthesis}
  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 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 — 1 There is also a syntax rule for formal-parameter-section in **6.6.3.7.1**.

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 defined to be a parameter-identifier for the region that 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 defined to be a parameter-identifier for the region that 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 — 2 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 type possessed by the formal-parameter shall be one that is permitted as the component-type of a file-type (see **6.4.3.5**). 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 type possessed by the actual-parameters shall be the same as that denoted by the type-identifier of the variable-parameter-specification, 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 that is the selector of a variant-part. An actual variable parameter shall not denote a component of a variable where that variable 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 formal-parameter-list, if any, closest-contained by the formal-parameter-section and the formal-parameter-list, if any, that defines the formal-parameters of the procedure denoted by the actual-parameter shall be congruous, or neither formal-parameter-list shall occur. 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 formal-parameter-list, if any, closest-contained by the formal-parameter-section and the formal-parameter-list, if any, that defines the formal-parameters of the function denoted by the actual-parameter shall be congruous, or neither formal-parameter-list shall occur. The result-type closest-contained by the formal-parameter-section shall denote the same type as the result type of the function. 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 following statements 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 either both value-conformant-array-specifications or both variable-conformant-array-specifications; and in both cases the conformant-array-parameter-specifications contain the same number of parameters and equivalent conformant-array-schemas. Two conformant-array-schemas shall be equivalent if all of the following four statements are true.

   1) There is a single index-type-specification 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 For the status of item e) above see **5.1 a)**, **5.1 b)**, **5.1 c)**, **5.2 a)**, and **5.2 b)**.

6.6.3.7 Conformant array parameters

NOTE — For the status of this subclause see **5.1 a)**, **5.1 b)**, **5.1 c)**, **5.2 a)**, and **5.2 b)**.

6.6.3.7.1 General

The occurrence of an identifier in the identifier-list contained by 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. A variable-identifier so defined shall be designated a *conformant-array-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 specified by that formal-parameter-list.

```
formal-parameter-section > conformant-array-parameter-specification .

conformant-array-parameter-specification = value-conformant-array-specification
                                         | variable-conformant-array-specification .

value-conformant-array-specification = identifier-list ':' conformant-array-schema .

variable-conformant-array-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 .

factor > bound-identifier .

bound-identifier = identifier .
```

NOTE — 1 There are also syntax rules for formal-parameter-section in **6.6.3.1** and for factor in **6.7.1.**

If a conformant-array-schema closest-contains a conformant-array-schema, then an abbreviated form of definition shall be permitted. In the abbreviated form, a single semicolon 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
```

Within the activation of the block, applied occurrences of the first identifier of an index-type-specification shall denote the smallest value specified by the corresponding index-type (see **6.6.3.8**) possessed by the actual-parameter, and applied occurrences of the second identifier of the index-type-specification shall denote the largest value specified by that index-type.

NOTE — 2 The object denoted by a bound-identifier is neither a constant nor a variable.

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 the fixed-component-type of the conformant-array-parameters defined in the conformant-array-parameter-specification and that 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 contained by the conformant-array-parameter-specification. The type denoted by the type-identifier contained by the conformant-array-schema contained by a conformant-array-parameter-specification shall be designated the *fixed-component-type* of the conformant-array-parameters defined by that conformant-array-parameter-specification.

NOTE — 3 The type possessed by the formal-parameter cannot be a string-type (see **6.4.3.2**) because it is not denoted by an array-type.

6.6.3.7.2 Value conformant arrays

The identifier-list in a value-conformant-array-specification shall be a list of value conformant arrays. Each actual-parameter corresponding to a value formal-parameter shall be an expression. The value of the expression shall be attributed before activation of the block to an auxiliary variable that the program does not otherwise contain. The type possessed by this variable shall be the same as that possessed by the expression. This variable 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. The fixed-component-type of a value conformant array shall be one that is permitted as the component-type of a file-type.

If the actual-parameter contains an occurrence of a conformant-array-parameter, then for each occurrence of the conformant-array-parameter contained by the actual-parameter, either

a) the occurrence of the conformant-array-parameter shall be contained by a function-designator contained by the actual-parameter; or

b) the occurrence of the conformant-array-parameter shall be contained by an indexed-variable contained by the actual-parameter, such that the type possessed by that indexed-variable is the fixed-component-type of the conformant-array-parameter.

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

6.6.3.7.3 Variable conformant arrays

The identifier-list in a variable-conformant-array-specification shall be a list of variable conformant arrays. The actual-parameter shall be a variable-access. 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-parameter shall not denote a component of a variable where that variable possesses a type that is designated packed.

6.6.3.8 Conformability

NOTE — 1 For the status of this subclause see **5.1 a)**, **5.1 b)**, **5.1 c)**, **5.2 a)**, and **5.2 b)**.

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 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 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 — 2 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**).

At any place where the rule of conformability is used, 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

The required procedure-identifiers and function-identifiers and the corresponding required procedures and functions shall be as specified in **6.6.5**, **6.6.6**, and **6.9**.

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 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, while f (within an assertion) shall denote the variable after the operation, and similarly for f0↑ and f↑.

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

NOTE — In order to facilitate interactive terminal input and output, the procedure **get** (and other input procedures) should be performed at the latest opportunity, and the procedure **put** (and other output procedures) should be performed at the first opportunity. This technique has been called 'lazy I/O'.

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 possesses the type denoted by the required type-identifier **text** and if f0.L ~f0.R is not empty and if (f0.L ~f0.R).last is not 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 is 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.

read

Let f denote a file-variable and v_1,...,v_n denote variable-accesses (n>=2); then the procedure-statement read(f,v_1,...,v_n) shall access the file-variable and establish a reference to that file-variable for the remaining execution of the statement. The execution of the statement shall be equivalent to

```
begin read(ff,v1); read(ff,v2,...,vn) end
```

where ff denotes the referenced file-variable.

Let f be a file-variable and v be a variable-access; then the procedure-statement read(f,v) shall access the file-variable and establish a reference to that file-variable for the remaining execution of the statement. The execution of the statement shall be equivalent to

```
begin v := ff↑; get(ff) end
```

where ff denotes the referenced file-variable.

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

Let f denote a file-variable and e_1,...,e_n denote expressions (n>=2); then the procedure-statement write(f,e_1,...,e_n) shall access the file-variable and establish a reference to that file-variable for the remaining execution of the statement. The execution of the statement shall be equivalent to

```
begin write(ff,e1); write(ff,e2,...,en) end
```

where ff denotes the referenced file-variable.

Let f be a file-variable and e be an expression; then the procedure-statement write(f,e) shall access the file-variable and establish a reference to that file-variable for the remaining execution of the statement. The execution of the write statement shall be equivalent to

```
begin ff↑ := e; put(ff) end
```

where ff denotes the referenced file-variable.

NOTES

1 The required procedures **read, write, readln, writeln,** and **page,** as applied to textfiles, are described in **6.10**.

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.

3 A consequence of the definition of **read** and **write** is that the non-file parameters are evaluated in a left-to-right order.

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,c_1,...,c_n)

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 possessed by p and shall have nested variants that correspond to the case-constants c_1,...,c_n. 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 is 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,c_1,...,c_n).

dispose(q,k_1,...,k_m)

shall remove the identifying-value denoted by the expression q from the pointer-type of q. The case-constants k_1,...,k_m shall be listed in order of increasing nesting of the variant-parts. It shall be an error unless the variable had been created using the form new(p,c_1,...,c_n) and m is equal to n. It shall be an error if the variants in the variable identified by the pointer value of q are different from those specified by the values denoted by the case-constants k_1,...,k_m.

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 **6.6.3.2** and **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

In the statement pack(a,i,z) and in the statement unpack(z,a,i) the following shall hold: a and z shall be variable-accesses; a shall possess an array-type not designated packed; z shall possess an array-type designated packed; the component-types of the types of a and z shall be the same; and the value of the expression i shall be assignment-compatible with the index-type of the type of a.

Let j and k denote auxiliary variables that the program does not otherwise contain and that have the type that is the index-type of the type of z and a, respectively. Let u and v denote the smallest and largest values of the index-type of the type of z. Each of the statements pack(a,i,z) and unpack(z,a,i) shall establish references to the variables denoted by a and z for the remaining execution of the statements; let aa and zz, respectively, denote the referenced variables within the following sentence. Then the statement pack(a,i,z) shall be equivalent to

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

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

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

NOTE — Errors will arise if the references cannot be established, if one or more of the values attributed to j is not assignment-compatible with the index-type of the type of a, or if an evaluated array component is undefined.

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. The result shall be as shown in table 2.

Table 2 — Arithmetic function results

Function	Result
abs(x)	absolute value of x
sqr(x)	square of x It shall be an error if such a value does not exist.
sin(x)	sine of x, where x is in radians
cos(x)	cosine of x, where x is in radians
exp(x)	base of natural logarithms raised to the power x
ln(x)	natural logarithm of x, if x is greater than zero It shall be an error if x is not greater than zero.
sqrt(x)	non-negative square root of x, if x is not negative It shall be an error if x is negative.
arctan(x)	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 $0 \leq x - \text{trunc}(x) < 1$; otherwise, $-1 < x - \text{trunc}(x) \leq 0$. 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 that 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, it shall be true that

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**) and the program shall contain a program-parameter-list containing an identifier with the spelling input. 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**) and the program shall contain a program-parameter-list containing an identifier with the spelling input. When eoln(f) is activated, it shall be an error if f is undefined or if eof(f) is true; otherwise, the function shall yield the value true if f.R.first is an end-of-line component (see **6.4.3.5**); otherwise, false.

6.7 Expressions

6.7.1 General

An expression shall denote a value. The use of a variable-access as a factor shall denote the value, if any, attributed to the variable accessed thereby. When a factor is used, it shall be an error if the variable denoted by a variable-access of the factor is undefined. 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.

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

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

term = factor { multiplying-operator factor } .

factor > variable-access | unsigned-constant | function-designator
       | set-constructor | '(' expression ')' | 'not' factor .
```

NOTE — 1 There is also a syntax rule for factor in **6.6.3.7.1**.

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

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

member-designator = expression [ '..' expression ] .
```

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

A set-constructor shall denote a value of a set-type. The set-constructor *[ ]* shall denote the 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 be the value of x. The member-designator x..y, where x and y are expressions, shall denote zero or more members that shall be the values of the base-type in the closed interval from the value of x to the value of y. The order of evaluation of the expressions of a member-designator shall be implementation-dependent. The order of evaluation of the member-designators of a set-constructor shall be implementation-dependent.

NOTES

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

3 The set-constructor [ ] does not have a single type, but assumes a suitable type to satisfy the assignment-compatibility rules, or the compatibility rules for operators, if possible.

Examples:

a) Factors:

```
                    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, 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 3 and 4 respectively.

NOTE — 1 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)

Table 3 — Dyadic arithmetic operations

Operator	Operation	Type of operands	Type of result
+	Addition	integer-type or real-type	integer-type if both operands are of integer-type otherwise real-type
–	Subtraction	integer-type or real-type	
*	Multiplication	integer-type or 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 4 — 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

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.

NOTE — 2 Only for i >= 0 and j > 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 integer-to-real conversion, of the real arithmetic operators and of the required real 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. The Boolean operators or, and, and not shall denote respectively the logical operations of disjunction, conjunction, and negation.

Table 5 — Set operations

Operator	Operation	Type of operands	Type of result
+	Set union	The same unpacked-canonical-set-of-T-type or packed-canonical-set-of-T-type (see **6.7.1**)	Same as operands
–	Set difference		
*	Set intersection		

Table 6 — Relational operations

Operator	Type of operands	Type of result
= <>	Any simple-type, pointer-type, string-type, unpacked-canonical-set-of-T-type or packed-canonical-set-of-T-type	Boolean-type
< >	Any simple-type or string-type	Boolean-type
<= >=	Any simple-type, string-type, unpacked-canonical-set-of-T-type or packed-canonical-set-of-T-type	Boolean-type
in	Left operand: any ordinal-type T right operand: the unpacked-canonical-set-of-T-type or packed-canonical-set-of-T-type	Boolean-type

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 5.

Where x denotes a value of the ordinal-type T and u and v are operands of a canonical-set-of-T-type, it shall be true for all x that

— x is a member of the value u+v if and only if it is a member of the value of u or a member of the value of v;

— x is a member of the value u–v if and only if it is a member of the value of u and not a member of the value of v;

— x is a member of the value u*v if and only if it is a member of the value of u and a member of the value of v.

6.7.2.5 Relational operators

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

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

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

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

When the relational-operators = , <> , < , > , <=, and >= are used to compare operands of compatible string-types (see **6.4.3.2**), they shall denote the lexicographic relations defined below. This lexicographic ordering shall impose a total ordering on values of a string-type.

If s1 and s2 are two values of compatible string-types and n denotes the number of components of the 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]
```

The definitions of operations >, <>, <=, and >= are derived from the definitions of = and <.

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 specify the activation of the block of the function-block associated with the function-identifier of the function-designator and shall yield the value of the result of the activation upon completion of the algorithm of the activation; it shall be an error if the result is undefined upon completion of the algorithm.

NOTE — When a function activation is terminated by a goto-statement (see **6.8.2.4**), the algorithm of the activation does not complete (see **6.2.3.2 a)**), and thus there is no error if the result of the activation is undefined.

If the function has any formal-parameters, there shall be an actual parameter-list in the function-designator. The actual-parameter-list shall be the 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-parameters 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.

NOTE — 1 A statement may be prefixed by a label.

A label, if any, of a statement S shall be designated as *prefixing* S. The label shall be permitted 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 — 2 A goto-statement within a block may refer to a label in an enclosing block, provided that the label prefixes a 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 activation result that is denoted by the function-identifier of the assignment-statement; the value shall be assignment-compatible with the type possessed, respectively, by the variable or by the activation result. The function-block associated (see **6.6.2**) with the function-identifier of an assignment-statement shall contain the assignment-statement.

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

The variable-access shall establish a reference to the variable during the execution of the assignment-statement. The order of establishing the reference to the variable and evaluating the expression shall be implementation-dependent.

The state of a variable or activation result when the variable or activation result 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 for an activation result or a variable that does not possess a structured-type.

Examples:

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

6.8.2.3 Procedure-statements

A procedure-statement shall specify the activation of the 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 the 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-parameters 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.

The procedure-identifier in a procedure-statement containing a read-parameter-list shall denote the required procedure **read**; the procedure-identifier in a procedure-statement containing a readln-parameter-list shall denote the required procedure **readln**; the procedure-identifier in a procedure-statement containing a write-parameter-list shall denote the required procedure **write**; the procedure-identifier in a procedure-statement containing a writeln-parameter-list shall denote the required procedure **writeln**.

```
procedure-statement = procedure-identifier ( [ actual-parameter-list ]
                      | read-parameter-list | readln-parameter-list
                      | write-parameter-list | writeln-parameter-list ) .
```

Examples:

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

NOTE — The fourth example is not for level 0.

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 exceptions a) or b) 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 shall specify 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 immediately 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; otherwise, it shall be an error.

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

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

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

case-index = expression .

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

6.8.3.6 Repetitive-statements

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

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

6.8.3.7 Repeat-statements

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

The 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

```
 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 denoted by 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 initial-value and the final-value shall be assignment-compatible with the type possessed by the control-variable if the statement of the for-statement is executed. After a for-statement is executed, other than being left by a goto-statement, the control-variable shall be undefined. Neither a for-statement nor any procedure-and-function-declaration-part of the block that closest-contains a for-statement shall contain a statement threatening the variable denoted by the control-variable of the for-statement.

A statement S shall be designated as *threatening* a variable V if one or more of the following statements is true.

a) S is an assignment-statement and V is denoted by the variable-access of S.

b) S contains an actual variable parameter that denotes V.

c) S is a procedure-statement that specifies the activation of the required procedure **read** or the required procedure **readln**, and V is denoted by variable-access of a read-parameter-list or readln-parameter-list of S.

d) S is a statement specified using an equivalent program fragment containing a statement threatening V.

Apart from the restrictions imposed by these requirements, the for-statement

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

shall be equivalent to

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

and the for-statement

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

shall be equivalent to

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

where temp1 and temp2 denote auxiliary variables that 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 } .

field-designator-identifier = identifier .

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 the 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 that is the statement of the with-statement; each applied occurrence of a field-designator-identifier shall denote that component of the record-variable that 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
```

Example:

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

has the same effect on the variable date as

```
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 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**, and the program shall contain a program-parameter-list containing an identifier with the spelling input.

The requirements of this subclause shall apply for the procedure read when applied to a textfile; therein, f shall denote the textfile. The effects of applying read(f,v) to the textfile f shall be defined by pre-assertions and post-assertions within the requirements of **6.6.5.2**. The pre-assertion of read(f,v) shall be the pre-assertion of get(f). Let t denote a sequence of components having the char-type; let r, s, and u each denote a value of the sequence-type defined by the structure of the type denoted by **text**; if u = S( ), then let t = S( ); otherwise, let u.first = end-of-line; let w = f0↑ or w = f0.R.first, where the decision as to which shall be implementation-dependent; and let r ~s ~t ~u = w ~f0.R.rest. The post-assertion of read(f,v) shall be

(f.M = f0.M) and (f.L ~ f.R = f0.L ~ f0.R) and (f.R = t ~ u) and
(if f.R = S( ) then (f↑ is totally-undefined) else (f↑ = f.R.first)).

NOTE — 1 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.

a) For n>=1, read(f,v_1,...,v_n) shall access the textfile and establish a reference to that textfile for the remaining execution of the statement; each of v_1,...,v_n shall be a variable-access possessing a type that is the real-type, is a string-type, or is compatible with the char-type or with the integer-type. For n>=2, the execution of read(f,v_1,...,v_n) shall be equivalent to

```
begin read(ff,v1); read(ff,v2,...,vn) end
```

where ff denotes the referenced textfile.

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

```
begin v := ff↑; get(ff) end
```

where ff denotes the referenced textfile.

NOTE — 2 To satisfy the post-assertions of **get** and of read(f,v) requires r = S( ) and length(s) = 1.

c) If v is a variable-access possessing the integer-type (or subrange thereof), read(f,v) shall satisfy the following requirements. No component of s shall equal end-of-line. The components of r, if any, shall each, and (s ~t ~u).first shall not, equal either the char-type value space or end-of-line. Either s shall be empty or s shall, and s ~S(t.first) shall not, form a signed-integer according to the syntax of **6.1.5**. It shall be an error if s is empty. The value of the signed-integer thus formed shall be assignment-compatible with the type possessed by v and shall be attributed to v.

NOTE — 3 The sequence r represents any spaces and end-of-lines to be skipped, and the sequence s represents the signed-integer to be read.

d) If v is a variable-access possessing the real-type, read(f,v) shall satisfy the following requirements. No component of s shall equal end-of-line. The components of r, if any, shall each, and (s ~t ~u).first shall not, equal either the char-type value space or end-of-line. Either s shall be empty or s shall, and s ~S(t.first) shall not, form a signed-number according to the syntax of **6.1.5**. It shall be an error if s is empty. The value denoted by the number thus formed shall be attributed to the variable v.

NOTE — 4 The sequence r represents any spaces and end-of-lines to be skipped, and the sequence s represents the number to be read.

6.9.2 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**, and the program shall contain a program-parameter-list containing an identifier with the spelling input.

Readln(f,v_1,...,v_n) shall access the textfile and establish a reference to that textfile for the remaining execution of the statement. The execution of the statement shall be equivalent to

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

where ff denotes the referenced textfile.

Readln(f) shall access the textfile and establish a reference to that textfile for the remaining execution of the statement. The execution of the statement shall be equivalent to

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

where ff denotes the referenced textfile.

NOTES

1 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.

2 Because the definition of **readln** makes use of **get**, the implementation-defined aspects of the post-assertion of **get** also apply (see **6.6.5.2**).

6.9.3 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**, and the program shall contain a program-parameter-list containing an identifier with the spelling 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**).

For n>=1, write(f,p_1,...,p_n) shall access the textfile and establish a reference to that textfile for the remaining execution of the statement. For n>=2, the execution of the statement shall be equivalent to

```
begin write(ff,p1); write(ff,p2,...,pn) end
```

where ff denotes the referenced textfile.

Write(f,p), where f denotes a textfile and p is a write-parameter, shall write a sequence of zero or more characters on the textfile f; for each character c in the sequence, the equivalent of

```
begin ff↑ := c; put(ff) end
```

where ff denotes the referenced textfile, shall be applied to the textfile f. The sequence of characters written shall be a representation of the value of the first expression in the write-parameter p, as specified in the remainder of this subclause.

NOTE — Because the definition of **write** includes the use of **put**, the implementation-defined aspects of the post-assertion of **put** also apply (see **6.6.5.2**).

6.9.3.1 Write-parameters

A write-parameter shall have one of the following forms

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

where e shall be an expression whose value is to be written on the file f and shall be of integer-type, real-type, char-type, Boolean-type, or a string-type, and where TotalWidth and FracDigits shall be expressions of integer-type whose values shall be 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 **6.9.3.4.2**).

6.9.3.2 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.
```

6.9.3.3 Integer-type

If e is of integer-type, the decimal representation of the value 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

a) 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).

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

6.9.3.4 Real-type

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

6.9.3.4.1 The floating-point representation

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

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

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

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

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

let ActWidth be the positive integer defined by

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

and let the non-negative number eWritten, the positive integer DecPlaces, and the integer ExpValue be defined by

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

6.9.3.4.2 The fixed-point representation

Write(f,e : TotalWidth : FracDigits) shall cause a fixed-point representation of the value of e to be written. Assume the functions TenPower, RealSize, and Truncate described in **6.9.3.4.1**;

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 RealSize ( eWritten ) < 1
  then IntDigits := 1
  else IntDigits := RealSize ( eWritten );
```

and let MinNumChars be the positive integer defined by

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

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

```
if TotalWidth >= MinNumChars,
    (TotalWidth – MinNumChars) spaces,
the character '–' if (e < 0.0) and (eWritten > 0.0),
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.

6.9.3.5 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-string `'True'` or `'False'` (see **6.9.3.6**), where the case of each letter is implementation-defined, with a field-width parameter of TotalWidth.

6.9.3.6 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 n-th characters of the value of e in that order.
if 1 <= TotalWidth <= n,
  the first through TotalWidth-th characters in that order.
```

6.9.4 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**, and the program shall contain a program-parameter-list containing an identifier with the spelling output.

Writeln(f,p_1,...,p_n) shall access the textfile and establish a reference to that textfile for the remaining execution of the statement. The execution of the statement shall be equivalent to

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

where ff denotes the referenced textfile.

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) and (f0.R = S( )).

post-assertion: (f.L = (f0.L ~ S(end-of-line))) 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, that 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).

6.9.5 The procedure page

It shall be an error if the pre-assertion required for writeln(f) (see **6.9.4**) 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**, and the program shall contain a program-parameter-list containing an identifier with the spelling 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-parameter-list ')'] .

program-parameter-list = identifier-list .

program-block = block .

The identifier of the program-heading shall be the program name. It shall have no significance within the program. The identifiers contained by the program-parameter-list shall be distinct and shall be designated *program-parameters*. Each program-parameter shall have a defining-point as a variable-identifier for the region that is 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 International Standard.

The execution of any action, operation, or function, defined within clause **6** to operate on a variable, shall be an error if the variable is a program-parameter and, as a result of the binding of the program-parameter, the execution cannot be completed as defined.

The occurrence of the required identifier **input** or **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 the required type-identifier **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.

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.

program t6p6p3p4 (output);
var globalone, globaltwo : integer;

procedure dummy;
 begin
 writeln('fail4')
 end { of dummy };

procedure p (procedure f(procedure ff; procedure gg); procedure g);
 var localtop : integer;
 procedure r;
  begin  { r }
  if globalone = 1 then
   begin
   if (globaltwo <> 2) or (localtop <> 1) then
    writeln('fail1')
   end
  else if globalone = 2 then
   begin
   if (globaltwo <> 2) or (localtop <> 2) then
    writeln('fail2')
   else
    writeln('pass')
   end
  else
   writeln('fail3');
  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  { of t6p6p3p4 }
globalone := 1;
globaltwo := 0;
p(q, dummy)
end.  { of t6p6p3p4 }
```

Annex A

(Informative)

Collected syntax

The nonterminal symbols pointer-type, program, signed-number, simple-type, special-symbol, and structured-type are only referenced by the semantics and are not used in the right-hand-side of any production. The nonterminal symbol program is the start symbol of the grammar.

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

6.7.3       actual-parameter-list = ‘(’ actual-parameter { ‘,’ actual-parameter } ‘)’ .

6.7.2.1     adding-operator = ‘+’ | ‘–’ | ‘or’ .

6.1.7       apostrophe-image = ‘"’ .

6.4.3.2     array-type = ‘array’ ‘[’ index-type { ‘,’ index-type } ‘]’ ‘of’ component-type .

6.5.3.2     array-variable = variable-access .

6.8.2.2     assignment-statement = ( variable-access | function-identifier ) ‘:=’ expression .

6.4.3.4     base-type = ordinal-type .

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

6.7.2.3     Boolean-expression = expression .

6.6.3.7.1   bound-identifier = identifier .

6.5.5       buffer-variable = file-variable ‘↑’ .

6.4.3.3     case-constant = constant .

6.4.3.3     case-constant-list = case-constant { ‘,’ case-constant } .

6.8.3.5     case-index = expression .

6.8.3.5     case-list-element = case-constant-list ‘:’ statement .

6.8.3.5     case-statement = ‘case’ case-index ‘of’ case-list-element
                             { ‘;’ case-list-element } [ ‘;’ ] ‘end’ .

6.1.7       character-string = ‘’’ string-element { string-element } ‘’’ .

6.4.3.2     component-type = type-denoter .

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

6.8.3.2     compound-statement = ‘begin’ statement-sequence ‘end’ .

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

6.6.3.7.1   conformant-array-parameter-specification = value-conformant-array-specification
                                                     | variable-conformant-array-specification .
```

6.6.3.7.1	conformant-array-schema = packed-conformant-array-schema | unpacked-conformant-array-schema .
6.3	constant = [sign] (unsigned-number | constant-identifier) | character-string .
6.3	constant-definition = identifier ‘=’ constant .
6.2.1	constant-definition-part = [‘const’ constant-definition ‘;’ { constant-definition ‘;’ }] .
6.3	constant-identifier = identifier .
6.8.3.9	control-variable = entire-variable .
6.1.1	digit = ‘0’ | ‘1’ | ‘2’ | ‘3’ | ‘4’ | ‘5’ | ‘6’ | ‘7’ | ‘8’ | ‘9’ .
6.1.5	digit-sequence = digit { digit } .
6.1.4	directive = letter { letter | digit } .
6.4.4	domain-type = type-identifier .
6.8.3.4	else-part = ‘else’ statement .
6.8.2.1	empty-statement = .
6.5.2	entire-variable = variable-identifier .
6.4.2.3	enumerated-type = ‘(’ identifier-list ‘)’ .
6.7.1	expression = simple-expression [relational-operator simple-expression] .
6.6.3.7.1	factor > bound-identifier .
6.7.1	factor > variable-access | unsigned-constant | function-designator | set-constructor | ‘(’ expression ‘)’ | ‘not’ factor .
6.5.3.3	field-designator = record-variable ‘.’ field-specifier | field-designator-identifier .
6.8.3.10	field-designator-identifier = identifier .
6.5.3.3	field-identifier = identifier .
6.4.3.3	field-list = [(fixed-part [‘;’ variant-part] | variant-part) [‘;’]] .
6.5.3.3	field-specifier = field-identifier .
6.4.3.5	file-type = ‘file’ ‘of’ component-type .
6.5.5	file-variable = variable-access .
6.8.3.9	final-value = expression .
6.4.3.3	fixed-part = record-section { ‘;’ record-section } .
6.8.3.9	for-statement = ‘for’ control-variable ‘:=’ initial-value (‘to’ | ‘downto’) final-value ‘do’ statement .
6.6.3.1	formal-parameter-list = ‘(’ formal-parameter-section { ‘;’ formal-parameter-section } ‘)’ .

6.6.3.1	formal-parameter-section > value-parameter-specification
	| variable-parameter-specification
	| procedural-parameter-specification
	| functional-parameter-specification .
6.6.3.7.1	formal-parameter-section > conformant-array-parameter-specification .
6.1.5	fractional-part = digit-sequence .
6.6.2	function-block = block .
6.6.2	function-declaration = function-heading ';' directive
	| function-identification ';' function-block
	| function-heading ';' function-block .
6.7.3	function-designator = function-identifier [actual-parameter-list] .
6.6.2	function-heading = 'function' identifier [formal-parameter-list] ':' result-type .
6.6.2	function-identification = 'function' function-identifier .
6.6.2	function-identifier = identifier .
6.6.3.1	functional-parameter-specification = function-heading .
6.8.2.4	goto-statement = 'goto' label .
6.5.4	identified-variable = pointer-variable '↑' .
6.1.3	identifier = letter { letter | digit } .
6.4.2.3	identifier-list = identifier { ',' identifier } .
6.8.3.4	if-statement = 'if' Boolean-expression 'then' statement [else-part] .
6.5.3.2	index-expression = expression .
6.4.3.2	index-type = ordinal-type .
6.6.3.7.1	index-type-specification = identifier '..' identifier ':' ordinal-type-identifier .
6.5.3.2	indexed-variable = array-variable '[' index-expression, { ',' index-expression } ']' .
6.8.3.9	initial-value = expression .
6.1.6	label = digit-sequence .
6.2.1	label-declaration-part = ['label' label { ',' label } ';'] .
6.1.1	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' .
6.7.1	member-designator = expression ['..' expression] .
6.7.2.1	multiplying-operator = '*' | '/' | 'div' | 'mod' | 'and' .
6.4.2.1	new-ordinal-type = enumerated-type | subrange-type .
6.4.4	new-pointer-type = '↑' domain-type .
6.4.3.1	new-structured-type = ['packed'] unpacked-structured-type .

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

6.4.2.1     ordinal-type = new-ordinal-type | ordinal-type-identifier .

6.4.2.1     ordinal-type-identifier = type-identifier .

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

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

6.4.1       pointer-type-identifier = type-identifier .

6.5.4       pointer-variable = variable-access .

6.6.3.1     procedural-parameter-specification = procedure-heading .

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

6.6.1       procedure-block = block .

6.6.1       procedure-declaration = procedure-heading ';' directive
                                  | procedure-identification ';' procedure-block
                                  | procedure-heading ';' procedure-block .

6.6.1       procedure-heading = 'procedure' identifier [ formal-parameter-list ] .

6.6.1       procedure-identification = 'procedure' procedure-identifier .

6.6.1       procedure-identifier = identifier .

6.8.2.3     procedure-statement = procedure-identifier ( [ actual-parameter-list ]
                                  | read-parameter-list | readln-parameter-list
                                  | write-parameter-list | writeln-parameter-list ) .

6.10        program = program-heading ';' program-block '.' .

6.10        program-block = block .

6.10        program-heading = 'program' identifier [ '(' program-parameter-list ')' ] .

6.10        program-parameter-list = identifier-list .

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

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

6.4.2.1     real-type-identifier = type-identifier .

6.4.3.3     record-section = identifier-list ':' type-denoter .

6.4.3.3     record-type = 'record' field-list 'end' .

6.5.3.3     record-variable = variable-access .

6.8.3.10    record-variable-list = record-variable { ',' record-variable } .

6.7.2.1     relational-operator = '=' | '<>' | '<' | '>' | '<=' | '>=' | 'in' .

6.8.3.7     repeat-statement = 'repeat' statement-sequence 'until' Boolean-expression .
```

6.8.3.6	repetitive-statement = repeat-statement | while-statement | for-statement .
6.6.2	result-type = simple-type-identifier | pointer-type-identifier .
6.1.5	scale-factor = [sign] digit-sequence .
6.7.1	set-constructor = '[' [member-designator { ',' member-designator }] ']' .
6.4.3.4	set-type = 'set' 'of' base-type .
6.1.5	sign = '+' | '−' .
6.1.5	signed-integer = [sign] unsigned-integer .
6.1.5	signed-number = signed-integer | signed-real .
6.1.5	signed-real = [sign] unsigned-real .
6.7.1	simple-expression = [sign] term { adding-operator term } .
6.8.2.1	simple-statement = empty-statement | assignment-statement | procedure-statement | goto-statement .
6.4.2.1	simple-type = ordinal-type | real-type-identifier .
6.4.1	simple-type-identifier = type-identifier .
6.1.2	special-symbol = '+' | '−' | '*' | '/' | '=' | '<' | '>' | '[' | ']' | '.' | ',' | ':' | ';' | '↑' | '(' | ')' | '<>' | '<=' | '>=' | ':=' | '..' | word-symbol .
6.8.1	statement = [label ':'] (simple-statement | structured-statement) .
6.2.1	statement-part = compound-statement .
6.8.3.1	statement-sequence = statement { ';' statement } .
6.1.7	string-character = one-of-a-set-of-implementation-defined-characters .
6.1.7	string-element = apostrophe-image | string-character .
6.8.3.1	structured-statement = compound-statement | conditional-statement | repetitive-statement | with-statement .
6.4.3.1	structured-type = new-structured-type | structured-type-identifier .
6.4.1	structured-type-identifier = type-identifier .
6.4.2.4	subrange-type = constant '..' constant .
6.4.3.3	tag-field = identifier .
6.4.3.3	tag-type = ordinal-type-identifier .
6.7.1	term = factor { multiplying-operator factor } .
6.4.1	type-definition = identifier '=' type-denoter .
6.2.1	type-definition-part = ['type' type-definition ';' { type-definition ';' }] .
6.4.1	type-denoter = type-identifier | new-type .