Extended Pascal
ISO 10206:1990

Use this information only for good; never for evil. Do not expose to fire. Do not operate heavy equipment after reading, may cause drowsiness. Do not read under the influence of alcohol (although there have been several unconfirmed reports that alcohol actually improves the readability). The standard is written in English. If you have trouble understanding a particular section, read it again and again and again... Sit up straight. Eat your vegatables. Do not mumble.

Acknowledgements

The efforts are acknowledged of all those who contributed to the work of WG2, and in particular:

Brian Wichmann
Convener

David Bustard	Harris Hall	John Reagan
Barry Byrne	Carsten Hammer	Paula Schwartz
Klaus Daessler	Tony Hetherington	Steven Siegfried
Norman Diamond	Steve Hobbs	Manfred Stadel
Bob Dietrich	David Joslin	Tom Turba
Ken Edwards	Jim Miner	Willem Wakker

The efforts are acknowledged of all those who contributed to the work of JPC, and in particular:

Thomas N. Turba Chairman X3J9	Michael Patrick Hagerty Chairman IEEE P770	John R. Reagan Secretary

Steve Adamczyk	Steven Hobbs	David L. Presberg
Jeffrey Allen	Albert A Hoffman	William C. Price
Edward Barkmeyer	Michael A. Houghtaling	Bruce Ravenel†
Beth Benoit	Robert C. Hutchins	David L. Reese
W. Ashby Boaz	Rosa C. Hwang	Mike Renfro
Jack Boudreaux	Scott Jameson	David C. Robbins
Jerry R. Brookshire	Janis Johnson	Richard H. Rosenbaum
A. Winsor Brown	Jay K Joiner	Lynne Rosenthal
Tom Bucken	David T. Jones	Thomas Rudkin
Thomas M. Burger	David Joslin	Steve Russell
David S. Cargo	Mel Kanner	Paula Schwartz
Joe Cointment	Leslie Klein	Evelyn Scott
Roger Cox	Dennis A. Kodimer	Wayne Sewell
Jean Danver	Ronald E. Kole	Steve Siegfried
Debra Deutsch	Bill Kraynek	Nancy Simmons
Bob Dietrich	Robert G. Lange	Dave Skinner
Jane Donaho	Charles Linett	Carol Sledge‡
Kenneth K. Edwards	David Lyman	Barry Smith
John Flores	Pat Mayekawa	Peter Steinfeld
Victor A. Folwarczny	Rainer F. McCown	Michael C. Stinson
Dennis Foster	Jim Miner	Prescott K. Turner
Thomas Giventer	Eugene N. Miya	Robert Tuttle
Hellmut Golde	Mark Molloy	Richard C. Vile, Jr
David N. Gray	Wes Munsil	Larry B. Weber
Paul Gregory	David Neal	David Weil
Ann Grossman	William Neuhauser	Brian Wichmann x
Harris Hall	Dennis Nicholson	Thomas R. Wilcox

Christopher J. Henrich	Mark Overgaard	Harvey Wohlwend
Tony Hetherington	Ted C. Park	Thomas Wolfe
Steven Hiebert	Donald D. Peckham	Kenneth M. Zemrowski‡
Ruth M. Higgins	David E. Peercy	
Charles R. Hill	Robert C. B. Poon	

† Past Chairman IEEE P770
‡ Past Chairman X3J9

Introduction

This International Standard provides an unambiguous and machine independent definition of the programming language Extended Pascal. Its purpose is to facilitate portability of Extended 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 reliable and efficient on then-available computers.

However, it has become apparent that Pascal has attributes that go far beyond those original goals. It is now being increasingly used commercially in the writing of system and application software. With this increased use, there has been an increased demand for and availability of extensions to ISO 7185:1983, *Programming languages - PASCAL*. Programs using such extensions attain the benefits of the extended features at the cost of portability with standard Pascal and with other processors supporting different sets of extensions. In the absence of a standard for an extended language, these processors have become increasingly incompatible. 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.

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 programme 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 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 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. (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.)

Following the decision that the first publication of a standard for the programming language Pascal would not contain extensions to the language, JPC prepared a project proposal to SPARC for an Extended Pascal Standard. When approved by X3 in November 1980, this proposal formed the charter for Project 345.

JPC immediately formed the Extension Task Group to receive all proposals for extensions to the Pascal language, developed the content of proposals so that they were in a form suitable for review by JPC, fairly and equitably reviewed all proposals in light of published JPC policy, and provided a liaison with the public in all matters concerning proposed extensions to the Pascal language.

X3 issued a press release on behalf of JPC in January 1980 to solicit extension proposals or suggestions from the general public. At this time, JPC had already prepared a list of priority extensions; public comment served to validate and supplement the priority list. Criteria for evaluating extensions were established and included machine independence, upward compatibility, conceptual integrity, rigorous definition, and existing practice as prime objectives. Extension proposals submitted by the public and by the JPC membership were developed and refined. JPC procedures guaranteed that proposals would be considered over at least two meetings, affording adequate time for review of the technical merits of each proposal.

By June of 1983, twelve extensions had been designated by JPC as candidate extensions and were published as a Candidate Extension Library. Ongoing work was described in Work in Progress, published with the Candidate Extension Library. This effort served as an interim milestone and an opportunity for the public to review the effort to date.

In 1984, BSI also started work on extensions to Pascal, with an initial aim of providing extensions in a few areas only. 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 to consider proposals from both ANSI and BSI in an international forum. Thereafter WG2 met at regular intervals to reconcile the national standardization activities in ANSI and BSI and to consider issues raised by the other experts participating in WG2.

The Work in Progress, along with other proposals subsequently received, continued its development until June 1986. The process of reconciling individual candidate extensions among themselves was begun in September 1984 and continued until June 1986. During this phase, conflicts between changes were resolved and each change was reconsidered. Working drafts of the full standard were circulated within JPC and WG2 to incorporate changes from each meeting.

The candidate extensions were then integrated into a draft standard that was issued for public review. The Public Comment Task Group (PCTG) was formed to respond to the public comments and recommend changes to the draft. To promote a unified response on each comment issue, PCTG included members from both WG2 and JPC. All responses and recommended changes required final approval by JPC and WG2. PCTG recommended several substantive changes that were subsequently approved as changes to the draft. These changes were incorporated and a new draft was produced for a second public review.

Project charter

The goal of JPC's Project 345 was to define an implementable, internationally acceptable Extended Pascal Standard.

This International Standard was to encompass those extensions found to be

a) compatible with ANSI/IEEE770X3.97-1983, American National Standard Programming Language Pascal, and

b) beneficial with respect to cost.

JPC's approved program of work included:

a) solicitation of proposals for extended language features;

b) the critical review of such proposals;

c) synthesis of those features found to be acceptable individually and which are mutually consistent into a working draft proposed standard;

d) interface with all interested standards bodies, both domestic and international;

e) submission of the working draft to ISO/TC97/SC22/WG2;

f) synthesis and submission of a draft proposed ANS consistent with any international standard developed;

g) review and correction of the dpANS in light of any comment received during Public Comment and/or Trial Use periods.

Technical development

Extended Pascal incorporates the features from ANSI/IEEE770X3.97-1983 and the following new features:

a) **Modularity and Separate Compilation**. Modularity provides for separately-compilable program components, while maintaining type security.

—Each module exports one or more interfaces containing entities (values, types, schemata, variables, procedures, and functions) from that module, thereby controlling visibility into the module.

—A variable may be protected on export, so that an importer may use it but not alter its value. A type may be restricted, so that its structure is not visible.

—The form of a module clearly separates its interfaces from its internal details.

—Any block may import one or more interfaces. Each interface may be used in whole or in part.

—Entities may be accessed with or without interface-name qualification.

—Entities may be renamed on export or import.

—Initialization and finalization actions may be specified for each module.

—Modules provide a framework for implementation of libraries and non-Pascal program components.

b) **Schemata**. A schema determines a collection of similar types. Types may be selected statically or dynamically from schemata.

—Statically selected types are used as any other types are used.

—Dynamically selected types subsume all the functionality of, and provide functional capability beyond, conformant arrays.

—The allocation procedure **new** may dynamically select the type (and thus the size) of the allocated variable.

—A schematic formal-parameter adjusts to the bounds of its actual-parameters.

—The declaration of a local variable may dynamically select the type (and thus the size) of the variable.

—The with-statement is extended to work with schemata.

—Formal schema discriminants can be used as variant selectors.

c) **String Capabilities**. The comprehensive string facilities unify fixed-length strings and character values with variable-length strings.

—All string and character values are compatible.

—The concatenation operator (+) combines all string and character values.

—Variable-length strings have programmer-specified maximum lengths.

—Strings may be compared using blank padding via the relational operators or using no padding via the functions **EQ, LT, GT, NE, LE**, and **GE**.

—The functions **length, index, substr,** and **trim** provide information about, or manipulate, strings.

—The substring-variable notation makes accessible, as a variable, a fixed-length portion of a string variable.

—The transfer procedures **readstr** and **writestr** process strings in the same manner that **read** and **write** process textfiles.

—The procedure **read** has been extended to read strings from textfiles.

d) **Binding of Variables**.

—A variable may optionally be declared to be bindable. Bindable variables may be bound to external entities (file storage, real-time clock, command lines, etc.). Only bindable variables may be so bound.

—The procedures **bind** and **unbind**, together with the related type **BindingType**, provide capabilities for connection and disconnection of bindable internal (file and non-file) variables to external entities.

—The function **binding** returns current or default binding information.

e) **Direct Access File Handling**.

—The declaration of a direct-access file indicates an index by which individual file elements may be accessed.

—The procedures **SeekRead**, **SeekWrite**, and **SeekUpdate** position the file.

—The functions **position**, **LastPosition**, and **empty** report the current position and size of the file.

—The update file mode and its associated procedure **update** provide in-place modification.

f) **File Extend Procedure**. The procedure **extend** prepares an existing file for writing at its end.

g) **Constant Expressions**. A constant expression may occur in any context needing a constant value.

h) **Structured Value Constructors**. An expression may represent the value of an array, record, or set in terms of its components. This is particularly valuable for defining structured constants.

i) **Generalized Function Results**. The result of a function may have any assignable type. A function result variable may be specified, which is especially useful for functions returning structures.

j) **Initial Variable State**. The initial state specifier of a type can specify the value with which variables are to be created.

k) **Relaxation of Ordering of Declarations**. There may be any number of declaration parts (labels, constants, types, variables, procedures, and functions) in any order. The prohibition of forward references in declarations is retained.

l) **Type Inquiry**. A variable or parameter may be declared to have the type of another parameter or another variable.

m) **Implementation Characteristics**. The constant **maxchar** is the largest value of type **char**. The constants **minreal**, **maxreal**, and **epsreal** describe the range of magnitude and the precision of real arithmetic.

n) **Case-Statement and Variant Record Enhancements**. Each case-constant-list may contain ranges of values. An **otherwise** clause represents all values not listed in the case-constant-lists.

o) **Set Extensions**.

—An operator (><) computes the set symmetric difference.

—The function **card** yields the number of members in a set.

—A form of the for-statement iterates through the members of a set.

p) **Date and Time**. The procedure **GetTimeStamp** and the functions **date** and **time**, together with the related type **TimeStamp**, provide numeric representations of the current date and time and convert the numeric representations to strings.

q) **Inverse Ord**. A generalization of **succ** and **pred** provides an inverse ord capability.

r) **Standard Numeric Input**. The definition of acceptable character sequences read from a textfile includes all standard numeric representations defined by ISO 6093.

s) **Nondecimal Representation of Numbers**. Integer numeric constants may be expressed using bases two through thirty-six.

t) **Underscore in Identifiers**. The underscore character (_) may occur within identifiers and *is significant* to their spelling.

u) **Zero Field Widths**. The total field width and fraction digits expressions in write parameters may be zero.

v) **Halt**. The procedure **halt** causes termination of the program.

w) **Complex Numbers**.

—The simple-type **complex** allows complex numbers to be expressed in either Cartesian or polar notation.

—The monadic operators + and - and dyadic operators +, −, *, /, =, <> operate on complex values.

—The functions **cmplx**, **polar**, **re**, **im**, and **arg** construct or provide information about complex values.

—The functions **abs**, **sqr**, **sqrt**, **exp**, **ln**, **sin**, **cos**, **arctan** operate on complex values.

x) **Short Circuit Boolean Evaluation**. The operators **and_then** and **or_else** are logically equivalent to **and** and **or**, except that evaluation order is defined as left-to-right, and the right operand is not evaluated if the value of the expression can be determined solely from the value of the left operand.

y) **Protected Parameters**. A parameter of a procedure or a function can be protected from modification within the procedure or function.

z) **Exponentiation**. The operators ** and **pow** provide exponentiation of integer, real, and complex numbers to real and integer powers.

A) **Subrange Bounds**. A general expression can be used to specify the value of either bound in a subrange.

B) **Tag Fields of Dynamic Variables**. Any tag field specified by a parameter to the procedure **new** is given the specified value.

Extended Pascal incorporates the following feature at level 1 of this standard:

Conformant Arrays. Conformant arrays provide upward compatibility with level 1 of ISO 7185, *Programming languages - PASCAL*.

Technical reports

During the development of this International Standard, various proposals were considered but not incorporated due to consideration of time and other factors. Selected proposals may be published as Technical Reports.

Information technology — Programming languages – Extended Pascal

1 Scope

1.1

This International Standard specifies the semantics and syntax of the computer programming language Extended 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 Extended Pascal;

c) the method of activating the program-block or the set of commands used to control the environment in which an Extended Pascal program is transformed and executed;

d) the mechanism by which programs written in Extended 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 Dynamic-violation

A violation by a program of the requirements of this International Standard that a processor is permitted to leave undetected up to, but not beyond, execution of the declaration, definition, or statement that exhibits (see clause **6**) the dynamic-violation.

3.2 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 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 either a dynamic-violation or 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 dynamic-violations or 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.

3.3 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.

3.4 Implementation-defined

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

3.5 Implementation-dependent

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

Table 1 — Metalanguage symbols

<table>
<tr><th>Metasymbol</th><th>Meaning</th></tr>
<tr><td>=</td><td>Shall be defined to be</td></tr>
<tr><td>></td><td>Shall have as an alternative definition</td></tr>
<tr><td>|</td><td>Alternatively</td></tr>
<tr><td>.</td><td>End of definition</td></tr>
<tr><td>[x]</td><td>0 or 1 instance of x</td></tr>
<tr><td>{ x }</td><td>0 or more instances of x</td></tr>
<tr><td>(x | y)</td><td>Grouping: either of x or y</td></tr>
<tr><td>‘xyz’</td><td>The terminal symbol xyz</td></tr>
<tr><td>meta-identifier</td><td>A nonterminal symbol</td></tr>
</table>

3.6 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**.

The characters required to form Extended 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;

—the y_1, y_2,..., or y_n *closest-containing* an x: that y_i for some i in [1..n], closest-containing an x such that for all j in ([1..n]-[i]) if a y_j closest-contains that x then that y_j contains that y_i.

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.

NOTE — Some language constructs or concepts are not defined completely in a single subclause, but collectively in more than one subclause.

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 **6.7.3.6 e)**, **6.7.3.7**, and **6.7.3.8**, 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 or dynamic-violation, report the result of this determination to the user of the processor before the activation of the program-block, if any, and shall prevent activation of the program-block, if any;

f) treat each violation that is designated a dynamic-violation in at least one of the following ways

   1) the processor shall report the dynamic-violation or the possibility of the dynamic-violation 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;

   2) the processor shall report the dynamic-violation during execution of the program;

   and if a dynamic-violation is reported during execution of the program, the processor shall terminate execution; if a dynamic-violation occurs within a declaration, definition, or statement, the execution of that declaration, definition, or statement shall not be completed;

   NOTE — 1 Dynamic-violations, like all violations except errors, must be detected.

g) treat each violation that is designated an error as either:

   1) a dynamic-violation; or

   2) there shall be a statement in an accompanying document that the error is not reported, and a note referencing each such treatment shall appear in a separate section of the accompanying document;

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

   NOTE — 2 This means that processing will continue up to or beyond execution of the program at the option of the user.

h) 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 Extended Pascal as specified by ISO/IEC 10206';

i) be able to process, in a manner similar to that specified for errors, any use of any such extension; and

j) be able to process, in a manner similar to that specified for errors, any use of an implementation-dependent feature.

NOTE — 3 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 standard, the compliance statement shall be:

⟨*This processor*⟩ complies with the requirements of level ⟨*number*⟩ of ISO/IEC 10206.

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 10206 with the following exceptions: ⟨*followed by a reference to, or a complete list of, the requirements of the 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 — 4 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 **6.7.3.6 e)**, **6.7.3.7**, and **6.7.3.8**;

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 conforms 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 Extended Pascal programs are classified into special-symbols, identifiers, remote-directives, interface-directives, implementation-directives, unsigned-numbers, extended-numbers, labels, and character-strings. The representation of any character (upper case or lower case, differences of font, etc.) occurring anywhere outside of a character-string (see **6.1.9**) 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' | 'and_then' | 'array' | 'begin' | 'bindable' | 'case'
            | 'const' | 'div' | 'do' | 'downto' | 'else' | 'end' | 'export'
            | 'file' | 'for' | 'function' | 'goto' | 'if' | 'import'
            | 'in' | 'label' | 'mod' | 'module' | 'nil' | 'not' | 'of'
            | 'only' | 'or' | 'or_else' | 'otherwise' | 'packed' | 'pow'
            | 'procedure' | 'program' | 'protected' | 'qualified'
            | 'record' | 'repeat' | 'restricted' | 'set' | 'then' | 'to'
            | 'type' | 'until' | 'value' | '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 **6.12**).

```
identifier = letter { [ underscore ] ( letter | digit ) } .

underscore = '_' .
```

NOTE — An identifier cannot begin or end with an underscore, nor can two underscores be adjacent.

Examples:

```
X
time
readinteger
WG2
AlterHeatSetting
```

```
GInqWsTran
DeviceDriverIdentificationHeader
DeviceDriverIdentificationBody
Trondheim_Hammer_Dance
```

6.1.4 Remote-directives

A remote-directive shall only occur in a procedure-declaration or a function-declaration. The remote-directive shall be the required remote-directive **forward** (see **6.7.1** and **6.7.2**).

remote-directive = directive .

directive = letter { [underscore] (letter | digit) } .

NOTE — Many processors provide, as an extension, the remote-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. When providing such an extension, a processor should enforce the rules of Extended Pascal pertaining to type compatibility.

6.1.5 Interface-directives

An interface-directive shall only occur in a module-heading of a module-declaration. The interface-directive shall be the required interface-directive **interface** (see **6.11.1**).

interface-directive = directive .

NOTE — A processor may provide, as an extension, the interface-directive **external**, which is used to specify that the module-block corresponding to the module-heading containing the interface-directive is in some form other than an Extended Pascal module-block (e.g., it is implemented in some other language). When providing such an extension, a processor should enforce the rules of Extended Pascal pertaining to type compatibility.

6.1.6 Implementation-directives

An implementation-directive shall only occur in a module-identification of a module-declaration. The implementation-directive shall be the required implementation-directive **implementation** (see **6.11.1**).

implementation-directive = directive .

6.1.7 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**).

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 } .

number = signed-number
| [sign] (digit-sequence '.' | '.' fractional-part) ['e' scale-factor] .

NOTE — 1 The meta-identifier *number* is only used in **6.10.1 d**).

Examples:

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

An extended-digit that is a digit shall denote a digit-value which shall be the number of predecessors of that digit in the syntactic definition of digit in **6.1.1**. An extended-digit that is a letter shall denote a digit-value which shall be greater by ten than the number of predecessors of that letter in the syntactic definition of letter in **6.1.1**. The unsigned-integer of an extended-number shall denote the radix of the extended-number; the radix shall be in the closed interval two through thirty-six. No extended-digit in an extended-number shall denote a digit-value that equals or exceeds the radix of the extended-number. An extended-number shall denote, in conventional positional notation with the specified radix, a value of integer-type in the closed interval 0 to **maxint** (see **6.4.2.2**).

extended-digit = digit | letter .

extended-number = unsigned-integer '#' extended-digit { extended-digit } .

Examples:

```
16#ff
8#377
```

```
32#100
13#42     { the answer to the ultimate question of life,
              the universe, and everything              }
```

NOTE — 2 The character # is regarded as identical to corresponding currency symbols that appear in some national variants of ISO 646.

6.1.8 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 .

6.1.9 Character-strings

A character-string containing a single string-element shall denote a value of the char-type (see **6.4.2.2**). A character-string containing other than a single string-element shall denote a value of the canonical-string-type (see **6.4.3.3.1**) with a length equal to the number of string-elements contained in the character-string.

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 = 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'
';'
''''
'Extended Pascal'
'THIS IS A STRING'
'Don''t think this is two strings'
```

6.1.10 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.

The substitution of a space for a comment shall not alter the meaning of a program.

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 a program text. There shall be at least one separator between any pair of consecutive tokens made up of identifiers, word-symbols, labels, extended-numbers, or unsigned-numbers. No separators shall occur within tokens.

6.1.11 Lexical alternatives

The representation for lexical tokens and separators given in **6.1.1** to **6.1.10**, except for the character sequences (* and *), shall constitute a *reference representation* for these tokens and separators.

To facilitate the use of Extended 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 alternative 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 **6.1.10**).

NOTE — 2 See also **1.2 f)**.

6.2 Blocks, scopes, activations, and states

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 a 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 = import-part
        { label-declaration-part
         | constant-definition-part
         | type-definition-part
         | variable-declaration-part
         | procedure-and-function-declaration-part }
        statement-part .

import-part = [ 'import' import-specification ';' { import-specification ';' } ] .

label-declaration-part = 'label' label { ',' label } ';' .

constant-definition-part = 'const' constant-definition ';' { constant-definition ';' } .

type-definition-part = 'type' ( type-definition | schema-definition ) ';'
                       { ( type-definition | schema-definition ) ';' } .

variable-declaration-part = 'var' variable-declaration ';' { variable-declaration ';' } .

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

A procedure-and-function-declaration-part shall not be immediately followed by another procedure-and-function-declaration-part.

NOTE — A procedure-and-function-declaration-part thus consists of a maximal sequence of procedure-declarations, function-declarations, and semicolons. See the discussion of the remote-directive **forward** in **6.7.1** and **6.7.2**.

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, with the exception of the identifier of a program-heading (see **6.12**).

6.2.2.2

Each defining-point shall have one or more *regions* that are parts of the program text, and a scope that is part or all of those regions. The region that is an interface (see **6.11.2**), however, shall not be a part of the program text and shall be disjoint from every other interface.

6.2.2.3

The region(s) of each defining-point are defined elsewhere (see **6.2.1**, **6.2.2.10**, **6.2.2.12**, **6.3**, **6.4.1**, **6.4.2.3**, **6.4.3.4**, **6.4.7**, **6.5.1**, **6.5.3.3**, **6.7.1**, **6.7.2**, **6.7.3.1**, **6.7.3.7.1**, **6.8.4**, **6.8.6.3**, **6.8.7.3**, **6.8.8.3**, **6.9.3.10**, **6.11.1**, **6.11.2**, **6.11.3**, and **6.12**).

6.2.2.4

The scope of each defining-point shall be its region(s) (including all regions enclosed by those regions) 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, the field-specifier of a field-designated-constant, the field-specifier of a record-function-access, the discriminant-specifier of a schema-discriminant, a field-identifier of a field-value, the field-identifier of a tag-field-identifier, the identifier-list of the program-parameter-list, the identifier-list of the module-parameter-list, or the import-qualifier of an import-specification shall be excluded from the enclosing scopes. The region that is the constant-identifier of a constant-name, the type-identifier of a type-name, the schema-identifier of a schema-name, the variable-identifier of a variable-name, the procedure-identifier of a procedure-name, or the function-identifier of a function-name shall be excluded from the enclosing scopes if the constant-name, type-name, schema-name, variable-name, procedure-name, or function-name, respectively, contains an imported-interface-identifier.

NOTE — Consider the variable-name i1.x (see **6.5.1**) constructed from an interface-identifier i1 and a variable-identifier x. The part of the program text occupied by this occurrence of x is the region that is excluded from enclosing scopes. This region thus cannot be occupied by any other identifier that would be legal in a variable-identifier position and that has a scope that otherwise would include the region occupied by x.

For example in:

```
procedure a;
import i1 qualified only (x);
var y : integer;
begin
   i1.x := ...
```

the construct i1.x is allowed but i1.y is disallowed.

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 unless both identifiers are imported identifiers and denote the same value, variable, procedure, function, schema, or type. In the case of imported type-identifiers, both identifiers shall also denote the same bindability and initial state (see **6.11.3**).

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.

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 two exceptions:

a) An identifier can have an applied occurrence as a type-identifier or schema-identifier contained by the domain-type of any new-pointer-types contained by the type-definition-part containing the defining-point of the type-identifier or schema-identifier.

b) An identifier can have an applied occurrence as a constant-identifier, type-identifier, schema-identifier, variable-identifier, procedure-identifier, or function-identifier contained by an export-list closest-contained by a module-heading containing the defining-point of the identifier.

6.2.2.10

Required identifiers that denote the required values, types, schemata, procedures, and functions shall be used as if their defining-points have a region enclosing the program (see **6.1.3**, **6.4.2.2**, **6.4.3.4**, **6.4.3.6**, **6.4.3.3.3**, **6.7.5**, **6.7.6**, and **6.10**).

NOTES

1 The required identifiers **input** and **output** are not included, since these denote variables (see **6.11.4.2**).

2 The required identifiers **StandardInput** and **StandardOutput** are not included, since these denote interfaces (see **6.11.4.2**).

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 defining occurrence 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.

6.2.2.12

Each defining-point that has as a region a module-heading shall also have as a region the module-block that is associated with that module-heading.

6.2.2.13

A module A shall be designated as *supplying* a module B if A supplies the module-heading or module-block of B. A module A shall be designated as *supplying* a main-program-block if the module supplies the block of the main-program-block. A module A shall be designated as *supplying* a module-heading, module-block, or block, B, either if B contains an applied occurrence of an interface-identifier having a defining occurrence contained by the module-heading of A, or if A supplies a module that supplies B.

No module shall supply its module-heading.

NOTE — A module-heading that exports an interface precedes any module-heading, module-block, or block that imports the interface, and a module-heading precedes its module-block (see **6.2.2.9**).

6.2.3 Activations

6.2.3.1

A variable-identifier having a defining-point within a variable-declaration-part, for the region that is a module-block (see also **6.2.2.12**) or a block shall be designated *local* to the module containing the module-block (see **6.11.1**) or to the block, respectively.

A procedure-identifier or function-identifier having a defining-point within a procedure-and-function-heading-part or a procedure-and-function-declaration-part, for a region that is a module-block or a block, shall be designated *local* to the module containing the module-block or to the block, respectively.

6.2.3.2

Each activation of a block or module shall contain

a) for the statement-part of the block, an algorithm, the completion of which shall terminate the activation (see also **6.9.2.4**);

b) for each defining-point of a label in a label-declaration-part of the block, a corresponding program-point (see **6.2.1**);

c) for each new-type closest-contained by the module-heading of the module, the module-block of the module, or the block, one or more corresponding types (see **6.4.1**);

d) for each schema-definition containing a formal-discriminant-part and closest-contained by the module-heading of the module, the module-block of the module, or the block, a corresponding schema (see **6.4.7**);

e) for each conformant-array-form closest-contained by the formal-parameter-list, if any, defining the formal-parameters of the block, a corresponding type (see **6.7.3.7.1**);

f) for each defining-point of a variable-identifier local to the block or module, a corresponding variable (see **6.5.1**);

g) for each defining-point of a variable-identifier that is a formal-parameter of the block, occurring within a value-parameter-specification or a value-conformant-array-specification, a corresponding variable (see **6.7.3.1**, **6.7.3.2**, **6.7.3.7.1**, and **6.7.3.7.2**);

h) for each defining-point of a variable-identifier that is a formal-parameter of the block, occurring within a variable-parameter-specification or a variable-conformant-array-specification, a reference to the corresponding variable (see **6.7.3.1**, **6.7.3.3**, **6.7.3.7.1**, and **6.7.3.7.3**);

i) for each defining-point of a procedure-identifier local to the block or module, a corresponding procedure with the procedure-block corresponding to the procedure-identifier, and the formal-parameters of that procedure-block (see **6.7.1**);

j) for each defining-point of a function-identifier local to the block or module, a corresponding function with the function-block corresponding to, and the type associated with, the function-identifier, and the formal-parameters of that function-block (see **6.7.2**);

k) if the block is a function-block, a variable called the *result* of the activation, possessing the type and initial state (see **6.7.2**) associated with the block of the function-block, and possessing the bindability that is nonbindable;

l) if the block is a main-program-block, each textfile required to be implicitly accessible (see **6.11.4.2**) by any procedure-statement or function-designator contained by the program containing the main-program-block;

m) a commencement (see **6.2.3.8**);

n) for the module, an initialization, which shall be specified by a statement: if an initialization-part occurs in the module-block of the module, then the statement of the initialization-part; otherwise, an empty-statement (see **6.11.1**); and

o) for the module, a finalization, which shall be specified by a statement: if a finalization-part occurs in the module-block of the module, then the statement of the finalization-part; otherwise, an empty-statement (see **6.11.1**).

NOTE — Each activation contains its own algorithm, program-points, types, schemata, variables, references, commencement, initialization, finalization, procedures, and functions, distinct from those of every other activation.

6.2.3.3

An activation of a procedure or a function shall be an activation of the block of the procedure-block of the procedure or of the 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.

6.2.3.4

A procedure-statement or function-designator contained in the algorithm, initialization, or finalization of an activation and specifying an activation of a block shall be designated the *activation-point* of the activation of the block.

6.2.3.5

Each variable contained by an activation of a block or module, unless it is a program-parameter or module-parameter or it is a formal-parameter of the block, shall be created in its initial state (see **6.2.3.2 k)** and **6.5.1**) within the commencement of the activation. Each variable contained by an activation of a block or module, unless it is a program-parameter or module-parameter, shall be created not bound to an external entity. The algorithm, program-points, types, schemata, variables, references, finalization, procedures, and functions, if any, contained by an activation shall exist until the termination of the activation.

6.2.3.6

An activation of a program-block shall consist of an activation of the main-program-block contained by the program-block and, for each module supplying (see **6.2.2.13**) the main-program-block, an activation of that module. The termination of the activations of both the main-program-block and those modules shall constitute the termination of the activation of the program-block.

The order of any two distinct commencements shall be implementation-dependent unless the order is specified by the following sentence. Within an activation of a program-block, for each module or main-program-block A and for each module B other than A, if B supplies A and A does not supply

B, then the commencement of the activation of B shall precede the commencement of the activation of A.

The completion of the finalization of an activation of a module shall terminate the activation.

The order of the action specified by the finalization of an activation and the termination of a distinct activation shall be implementation-dependent unless the order is specified by the following sentence. Within an activation of a program-block, for each module or main-program-block A and for each module B other than A, if B supplies A and A does not supply B, then the termination of the activation of A shall precede the action specified by the finalization of the activation of B.

6.2.3.7

An activation of the module-heading or module-block associated with a module shall be the same activation of the module. An activation of a main-program-block shall be the activation of the block of the main-program-block.

6.2.3.8

The *commencement* of an activation of either a module or a block shall contain the following events

a) for each formal value parameter of the block, an attribution of a value to the variable denoted within the activation by the formal-parameter (see **6.7.3.2**), and for each formal variable parameter of the block, an access to the actual-parameter (see **6.7.3.3**);

b) for each actual-discriminant-part or subrange-bound not contained by a schema-definition and closest-contained by the module-heading of the module, by the module-block of the module, or by the block, the corresponding evaluation of the actual-discriminant-part or subrange-bound, respectively (see **6.4.8**);

c) for each defining occurrence of a variable-identifier local to the module or the block, the corresponding creation of the variable corresponding to the variable-identifier (see **6.2.3.2 f)** and **6.2.3.5**); and

d) the action specified by the initialization of the activation of the module.

Within the commencement of an activation, any events specified by a) shall precede any events specified by b) and c), and the latter events shall precede any event specified by d).

Within the commencement of an activation, the order of any events specified by b) and c) shall be the same as the textual order of their respectively-corresponding actual-discriminant-parts or subrange-bounds and defining occurrences, with one exception: An event specified by b) shall precede an event specified by c) if the respectively-corresponding actual-discriminant-part or subrange-bound and defining occurrence are both contained by one variable-declaration.

NOTE — An evaluation specified by b) can evaluate a local variable only if its initial state is value-bearing.

The commencement of an activation of a block shall precede the algorithm of the activation.

The completion of the events specified by a), b), c), and d) within a commencement shall constitute completion of the commencement.

6.2.4 States

A type determines a set of *states*, each of which shall be either a *value-bearing* state or a *non-value-bearing* state, but not both. A value-bearing state determined by a type shall be said to *bear* a value,and the values borne by two distinct value-bearing states shall be distinct. A non-value-bearing state shall not bear a value. When describing a state, *undefined* shall be synonymous with non-value-bearing.

The states determined by a structured-type shall have the structure of the structured-type.

The set of states determined by any type shall contain a special non-value-bearing state designated *totally-undefined.*

For any type that is not an array-type, a record-type, or a file-type, the set of states shall contain only the totally-undefined state and, for each value determined by the type, a state bearing that value.

NOTE — 1 The set of states determined by an array-type, a record-type, or a file-type is specified in **6.4.3.2**, **6.4.3.4**, and **6.4.3.6**, respectively, together with **6.4.2.1**.

For an array-type, a record-type, or a file-type, each component of the totally-undefined state shall be the totally-undefined state of the component-type, and each component of a value-bearing state shall be a value-bearing state.

NOTES

2 For a structured-type, each undefined state shall have at least one component that is undefined.

3 For a pointer-type, the set of states is dynamic in that the states bearing identifying-values (see **6.4.4**) are created and destroyed by actions of the program. Every non-pointer type determines a static set of values, i.e., a set that does not change during the existence of the type.

A variable declared to possess a type shall always have one of the states determined by the type; the particular state of a variable that is not bound to an external entity at any point shall have been determined by the actions specified by the program.

A value borne by the state of a variable shall be said to be *attributed* to the variable; a variable having a non-value-bearing state shall be said to have no value *attributed* to the variable and shall also be designated *undefined.*

NOTE — 4 Each state of a variable when the variable does not have attributed to it a value specified by its type is *undefined.* If a variable possesses a structured-type, the state of the variable when every component of the variable is totally-undefined is *totally-undefined.* Totally-undefined is synonymous with undefined for a variable that does not possess a structured-type.

Causing a variable to have the state bearing a value shall be described as *attributing* the value to the variable.

NOTE — 5 Subclauses that specify attribution (or de-attribution) of a value to a variable are: **6.5.3.3**, **6.6**, **6.7.3.2**, **6.7.3.7.2**, **6.7.5.2**, **6.7.5.3**, **6.7.5.4**, **6.7.5.5**, **6.7.5.6**, **6.7.5.8**, **6.7.6.7**, **6.9.2.2**, **6.9.3.9**, **6.10**, **6.11.4.2**. In some of these subclauses the attribution is implicit.

The *initial state* denoted by a type-denoter shall be a state determined by the type denoted by the type-denoter (see **6.6**).

6.3 Constants

6.3.1 General

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

constant-definition = identifier '=' constant-expression .

constant-identifier = identifier .

constant-name = [imported-interface-identifier '.'] constant-identifier .

A constant-name shall denote the value denoted by the constant-identifier of the constant-name.

The occurrence of an imported-interface-identifier in a constant-name shall be the defining-point of each imported constant-identifier associated with the imported-interface-identifier for the region that is the constant-identifier of the constant-name.

The occurrence of an identifier in a constant-definition of a constant-definition-part of a block, a module-heading, or a module-block shall constitute its defining-point as a constant-identifier for the region that is the block, the module-heading, or the module-block, respectively. A constant-expression in a constant-definition shall not contain an applied occurrence of the identifier in the constant-definition.

Each applied occurrence of the identifier in the constant-definition shall be a constant-identifier. Within an activation of the block, the module-heading, or the module-block, all applied occurrences of that identifier shall denote the value denoted by the constant-expression of the constant-definition. The required constant-identifiers shall be as specified in **6.4.2.2**.

NOTE — Constants of pointer-types are allowed, but they can only denote the value NIL.

6.3.2 Example of a constant-definition-part

NOTE — The type-identifiers *sieve, vector, quiver, PunchedCard,* and *subpolar* are defined in **6.4.10**.

```
const
unity = 1.0;
third = unity/3.0;                      { see 6.8.2 }
SmallPrimes = sieve[2,3,5,7,11,13,17,19];  { see 6.8.7.4 }
limit = 43;
ZeroVector = vector[1..limit: 0.0];       { see 6.8.7.2 }
UnitVector = vector[1: unity otherwise 0];
ZeroQuiver = quiver[otherwise ZeroVector];
```

```
BlankCard = PunchedCard[1..80: ' '];
blank = ' ';
Unit = subpolar[r:1; theta:0.0];           { see 6.8.7.3 }
Unit_Distance = Unit.r;                    { see 6.8.8.3 }
Origin = subpolar[r,theta:0.0];
thrust = 5.3; theta = -2.0; warp = subpolar[r:thrust;theta:theta];
column1 = BlankCard[1];                    { see 6.8.8.2 }
MaxMatrix = 39;
pi = 4 * arctan(1);
hex_string = '0123456789ABCDEF';
hex_digits = hex_string[1..10];           { see 6.8.8.4 }
hex_alpha = hex_string[index(hex_string,'A')..index(hex_string,'F')];
mister = 'Mr.';
```

6.4 Types and schemata

6.4.1 Type-definitions

A type-definition shall introduce an identifier to denote a type, bindability, and initial state (see **6.6**). *Bindability*, the quality of either being bindable or being nonbindable, but not both, shall be possessed by every variable. Type shall be an attribute that is possessed by every value and every variable. Within an activation of a block, module-heading, or module-block, closest-containing a new-type, the new-type shall denote one corresponding type and initial state if the new-type is not contained by a schema-definition (see **6.4.7**) and shall denote one or more mutually distinct corresponding types and initial states otherwise. Each type contained by an activation and corresponding to a new-type shall be distinct both from any type contained by any other activation, and from any type corresponding to any other new-type or conformant-array-form (see **6.2.3.2**).

type-definition = identifier '=' type-denoter .

type-denoter = ['bindable'] (type-name | new-type
| type-inquiry | discriminated-schema)
[initial-state-specifier] .

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

The occurrence of an identifier in a type-definition of a type-definition-part of a block, a module-heading, or a module-block shall constitute its defining-point for the region that is the block, the module-heading, or the module-block. Each applied occurrence of that identifier shall be a type-identifier. Within an activation of the block, the module-heading, or the module-block, all applied occurrences of that identifier shall denote the type, bindability, and initial state 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.

If the symbol bindable occurs in a type-denoter, the type-denoter shall denote the bindability that is bindable; otherwise, the type-denoter shall denote the bindability that is denoted by the type-name, the new-type, the type-inquiry, or the discriminated-schema of the type-denoter. The bindability denoted by a required type-identifier shall be nonbindable. A type-denoter denoting a restricted-type shall not contain the symbol bindable.

If an initial-state-specifier occurs in a type-denoter, the type-denoter shall denote the initial state that is denoted by the initial-state-specifier (see **6.6**); otherwise, the type-denoter shall denote the initial state that is denoted by the type-name, the new-type, the type-inquiry, or the discriminated-schema of the type-denoter. The initial state denoted by a required type-identifier shall be totally-undefined. A new-type shall denote the initial state denoted by the new-ordinal-type, the new-structured-type, the new-pointer-type, or the restricted-type of the new-type.

Types shall be classified as simple-types, restricted-types, structured-types, or pointer-types. The required type-identifiers and corresponding required types shall be as specified in **6.4.2.2**, **6.4.3.4**, and **6.4.3.6**. The required schema-identifier and the corresponding required schema shall be as specified in **6.4.3.3.3**.

simple-type-name = type-name .

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

array-type-name = type-name .

record-type-name = type-name .

set-type-name = type-name .

file-type-name = type-name .

pointer-type-name = type-name .

type-identifier = identifier .

type-name = [imported-interface-identifier '.'] type-identifier .

A type-name shall denote the type, bindability, and initial state denoted by the type-identifier of the type-name.

The occurrence of an imported-interface-identifier in a type-name shall be the defining-point of each imported type-identifier associated with the imported-interface-identifier for the region that is the type-identifier of the type-name.

A type-name shall be considered a simple-type-name, an array-type-name, a record-type-name, a set-type-name, a file-type-name, or a pointer-type-name, according to the type that it denotes.

A type shall be designated *protectable* unless

a) the type is either a file-type or a pointer-type, or

b) the type is a structured-type, and one or more of its component-type is not protectable.

NOTE — A file-type is not protectable since most operations on a file modify it in some way. A pointer-type is not protectable since the value of a pointer-type variable can be copied into another variable of the same type (possibly using type-inquiry), and then this value passed to the required procedure dispose. The required procedure dispose undefines all pointer variables denoting that identifying-value.

A type shall be designated *static* unless

a) the type is denoted by a subrange-type, and one or both subrange-bounds in the subrange-type denotes an expression that is not nonvarying, or

b) the type is produced from a schema, or

c) the type is denoted by an array-type or a file-type containing an index-type that denotes a type that is not static, or

d) the type is denoted by a structured-type containing any component whose type-denoter or selector-type denotes a type that is not static, or

e) the type is denoted by a set-type containing a base-type that denotes a type that is not static.

6.4.2 Simple-types

6.4.2.1 General

Each ordinal-type and the real-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-name, real-type-name, or complex-type-name shall denote an ordinal-type, the real-type, or the complex-type, respectively. A type-inquiry in an ordinal-type shall denote an ordinal-type.

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

ordinal-type = new-ordinal-type | ordinal-type-name
| type-inquiry | discriminated-schema .

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

ordinal-type-name = type-name .

real-type-name = type-name .

complex-type-name = type-name .

The *range-type* of an ordinal-type that is a subrange-type shall be the host-type (see **6.4.2.4**) of the subrange-type. The *range-type* of an ordinal-type that is not a subrange-type shall be the ordinal-type. A discriminated-schema in an ordinal-type shall denote an ordinal-type.

A new-ordinal-type shall denote the type, bindability, and initial state denoted by the subrange-type or the enumerated-type of the new-ordinal-type. The initial state denoted by an enumerated-type or a subrange-type shall be totally-undefined. The bindability denoted by an enumerated-type or a subrange-type shall be nonbindable.

6.4.2.2 Required simple-types and associated constants

NOTE — 1 Operators applicable to the required simple-types are specified in **6.8.3**.

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 **6.1.7** by signed-integer. The ordinal number of a value of integer-type shall be the value itself.

   The required constant-identifier **maxint** shall denote an implementation-defined value of integer-type. This value shall satisfy the following conditions.

   1) All integral values in the closed interval from -maxint to +maxint shall be values of the integer-type.

   2) Any monadic operation (see **6.8.3.2**) performed on an integer value in this interval shall be correctly performed according to the mathematical rules for integer arithmetic.

   3) Any dyadic integer operation (see **6.8.3.2**) 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.

   4) Any relational operation (see **6.8.3.5**) on two integer values in this same interval shall be correctly performed according to the mathematical rules for integer arithmetic.

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

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 implementation-defined approximations to an implementation-defined subset of the real numbers, denoted as specified in **6.1.7** by signed-real.

   NOTE — 2 The nature of the internal representation of values of real-type is not specified, and hence could be fixed-point, floating-point, or something quite different.

   Each of the required constant-identifiers **minreal**, **maxreal**, and **epsreal** shall denote an implementation-defined positive value of real-type. The values of **minreal** and **maxreal** shall be such that arithmetic in the set including the closed interval −maxreal to maxreal but excluding the two open intervals −minreal to zero and zero to minreal can be expected to work with reasonable approximations, but arithmetic outside this set cannot be expected to work with reasonable approximations. The value of **epsreal** shall be the result of subtracting 1.0 from the smallest value of real-type that is greater than 1.0.

   The results of integer-to-real conversion (see **6.4.6**), of the real arithmetic operators (see **6.8.3.2**), and of the required real functions (see **6.7.6**), shall be approximations to the corresponding mathematical results. The accuracy of this approximation shall be implementation-defined.

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.

   The required constant-identifier **maxchar** shall denote an implementation-defined value of char-type. The value of **maxchar** shall be the largest value of char-type.

   NOTE — 3 Char-type values possess properties that allow them to be used identically to string-type values of length 1. In particular, char-type values may be used to initialize a variable possessing a string-type (see **6.6**), used as the actual-parameter corresponding to a value parameter possessing a string-type (see **6.7.3.2**), used as the actual-parameter assigned to a conformant-actual-variable possessing a fixed-string-type and conforming to a value-conformant-array-specification (see **6.7.3.7.2**), assigned to a variable possessing a string-type (see **6.9.2.2**), written to a textfile (see **6.10.3.2**), used with the relational-operators (see **6.8.3.5**), and used with the string concatenation operator (see **6.8.3.6**). See also **6.4.5** and **6.4.6**.

e) *complex-type.* The required type-identifier **complex** shall denote the complex-type. The complex-type shall be a simple-type. The values shall be implementation-defined approximations to an implementation-defined subset of the complex numbers.

   NOTE — 4 The nature of the internal representation of values of complex-type is not specified, and hence could be rectangular, polar, or something quite different.

   The results of integer-to-complex and real-to-complex conversions (see **6.4.6**), of the complex arithmetic operators (see **6.8.3.2**), and of the required complex functions (see **6.7.6**), shall be approximations to the corresponding mathematical results. The accuracy of this approximation shall be implementation-defined.

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, module-heading, or module-block closest-containing the enumerated-type. Each applied occurrence of the identifier shall be a constant-identifier. Within an activation of the block, the module-heading, or the module-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. The identifier shall be designated a *principal identifier* of the value so denoted.

NOTES

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

2 While several identifiers may be known as principal identifiers of a given value (see **6.11.2** and **6.11.3**), there is no ambiguity, because each is defined for a different region, all have the same spelling, and all denote the same value.

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 subrange-bound of a subrange-type shall specify the smallest value. If both subrange-bounds of the subrange-type denote expressions that are nonvarying and do not contain a discriminant-identifier, the smallest value shall be less than or equal to the largest value, which shall be specified by the second subrange-bound of the subrange-type; otherwise, it shall be a dynamic-violation if the smallest value is not less than or equal to the largest value. The subrange-bounds shall be of compatible ordinal-types, and the range-type (see **6.4.2.1**) of the ordinal-types shall be designated the *host-type* of the subrange-type. An evaluation of a subrange-bound shall constitute evaluation of the expression of the subrange-bound (see **6.2.3.8**). The set of values determined by the subrange-type shall contain each value of the host-type not smaller than the smallest value and not larger than the largest value.

subrange-type = subrange-bound '..' subrange-bound .

subrange-bound = expression .

Examples:

```
1..100
```

```
-10..+10
red..green
'0'..'9'
```

6.4.2.5 Restricted-types

A restricted-type shall denote a type whose set of states is associated one-to-one with the states determined by another type, designated the *underlying-type* of the type denoted by the restricted-type. A type denoted by a restricted-type shall be designated *restricted*.

restricted-type = 'restricted' type-name .

The underlying-type of a restricted-type shall be the type denoted by the type-name of the restricted-type. The underlying-type of a type that is not restricted shall be the type, and each state shall be associated with itself. Attribution of a value of a type to a variable possessing the underlying-type of the type shall constitute the attribution of the associated value of the underlying-type. Attribution of a value of the underlying-type of a type to a variable possessing the type shall constitute the attribution of the associated value of the type. The bindability denoted by a restricted-type shall be nonbindable. The initial state denoted by a restricted-type shall be the state associated with the initial state denoted by the type-name of the restricted-type.

NOTE — A value of a restricted-type may be passed as a value parameter to a formal-parameter possessing its underlying-type (see **6.7.3.2**) or returned as the result of a function (see **6.9.2.2**). A variable of a restricted-type may be passed as a variable parameter to a formal-parameter possessing the same type or its underlying-type (see **6.7.3.3**). No other operations, such as accessing a component of a restricted-type value or performing arithmetic, are possible.

Example:

```
module widget_module;

export widgets = (widget, copy_widget, increment_widget, print_widget);

      { Access to the underlying-type (real_widget) of widget is
        controlled by not exporting it, thereby maintaining the
        privacy of widget. }

  type
    real_widget = record
                    f1 : integer;
                    f2 : real
                  end
                  value [f1:0; f2:0.0];

    widget = restricted real_widget;

       { widget can be thought of as having the same values
         and initial state as real_widget, but operations on
```

```
        it are restricted. }

  procedure copy_widget( source: real_widget; var target: real_widget );

  function increment_widget( w : real_widget ) : widget;

  procedure print_widget( var f: text; w : real_widget );

    { The parameters of these routines may accept actual-
      parameters that are of type widget or real_widget, but since
      real_widget is not exported and no variables of type real_widget
      are exported for possible use in a type-inquiry, a user of the
      interface can only pass actual-parameters of type widget. }
end;

  function increment_widget;
    var mycopy : real_widget;
    begin
  { Note that operations are performed on the underlying-type. }
      mycopy.f1 := w.f1 + 1;
      mycopy.f2 := w.f2 + 1.0;
  { An assignment from an underlying-type to a restricted-type. }
      increment_widget := mycopy;
    end;

  procedure copy_widget;
    begin
      target := source
    end;

  procedure print_widget;
    begin
  { Within the implementation of this module, the components of the
    actual-parameter are visible through its associated formal-
    parameter. The components of a variable of type widget are not
    visible outside the module, however, since the underlying-type
    is not exported. }

      writeln(f,w.f1,w.f2);
    end;
end.

program use_widgets( output );
  import widgets;

  var
```

```
    first, second: widget;

  begin
    write( output, 'First is initially ' ); print_widget( output, first );
    copy_widget( increment_widget( increment_widget( first ) ), second );
    write(output, 'Second is now '); print_widget( output, second );
    copy_widget( second, first );
    write(output, 'First is now '); print_widget( output, first );
  end.
```

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. A component of a state of a structured-type shall be a state.

structured-type = new-structured-type | structured-type-name .

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 states 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**, **6.7.3.3**, **6.7.3.7.3**, **6.7.5.4**, and **6.8.1**.

A new-structured-type shall denote the type, bindability, and initial state denoted by the unpacked-structured-type of the new-structured-type. An unpacked-structured-type shall denote the type and initial state denoted by the array-type, record-type, set-type, or file-type of the unpacked-structured-type. The bindability denoted by an unpacked-structured-type shall be nonbindable.

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, bindability, and initial state denoted by the type-denoter of the component-type of the array-type. The type-denoter of a component-type shall not closest-contain an initial-state-specifier (see **6.6**).

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

index-type = ordinal-type .

component-type = type-denoter .

Examples:

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

An array-type that specifies a sequence of two or more index-types shall be an abbreviated notation for an array-type specified to have as its index-type the first index-type in the sequence and to have a component-type that is an array-type specifying the sequence of index-types without the first 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.

Examples:

```
1)    array [Boolean] of array [1..10] of array [size] of real
      array [Boolean] of array [1..10, size] of real
      array [Boolean, 1..10, size] of real
      array [Boolean, 1..10] of array [size] of real

2)    packed array [1..10, 1..8] of Boolean
      packed array [1..10] of packed array [1..8] of Boolean
```

Let i denote a value of the index-type; let V_i denote a state 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 states of the array-type shall be the distinct k-tuples of the form

$$(V_m, \ldots, V_n).$$

NOTE — 2 A state of an array-type is value-bearing if and only if each of its component states is value-bearing. 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.

The ordinal-type of an index-type shall denote the bindability that is nonbindable.

6.4.3.3 String-types

6.4.3.3.1 General

A string-type shall be a fixed-string-type or a variable-string-type or the required type designated *canonical-string-type*. Each string-type value is a value of the canonical-string-type.

Each value of a string-type shall be structured as a one-to-one mapping from an index-domain to a set of components possessing the char-type. The index-domain shall be a finite set that is empty or that contains successive integers starting with 1.

The *length* of a string-type value shall be the number of members in its index-domain. The string-type value with length zero is designated the *null-string*.

The *length* of a char-type value shall be 1. The capacity of the char-type shall be 1.

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 — String-types possess properties that allow accessing a substring (see **6.5.6**) and reading from a textfile (see **6.10.1**). String-type values may be used as the actual-parameter corresponding to a value parameter possessing a string-type (see **6.7.3.2**), used as the actual-parameter assigned to a conformant-actual-variable possessing a fixed-string-type and conforming to a value-conformant-array-specification (see **6.7.3.7.2**), assigned to a variable possessing a string-type (see **6.9.2.2**), written to a textfile (see **6.10.3.6**), used with the relational-operators (see **6.8.3.5**), and used with the string concatenation operator (see **6.8.3.6**). See also **6.4.5** and **6.4.6**.

6.4.3.3.2 Fixed-string-types

A subrange-type shall be designated a *fixed-string-index-type* if and only if the expression in the first subrange-bound in the subrange-type is nonvarying (see **6.8.2**), does not contain a discriminant-identifier, and denotes the integer value 1. Any type designated packed and denoted by an array-type having as its index-type a denotation of a fixed-string-index-type and having as its component-type a denotation of the char-type, shall be designated a *fixed-string-type*.

The *capacity* of a fixed-string-type shall be the largest value of its index-type.

NOTES

1 A fixed-string-type possesses the properties of both an array-type and a string-type.

2 The length of all values of a particular fixed-string-type is equal to the capacity of the fixed-string-type.

Example:

```
packed array [1..5] of char { capacity 5, length 5 }
```

6.4.3.3.3 Variable-string-types

There shall be a schema (see **6.4.7**) that is denoted by the required schema-identifier **string**. The schema **string** shall have one formal discriminant denoted by the required discriminant-identifier **capacity**, which shall possess the integer-type. Each type derived from the schema **string** shall be designated a *variable-string-type*. Each tuple in the domain of the schema shall have one component that is a value of integer-type greater than zero, and the component shall be designated the *capacity* of the variable-string-type produced from the schema with the tuple. Each value of a variable-string-type shall be a string-type value with a length less than or equal to the capacity of the variable-string-type.

Example:

```
string(6)      { capacity 6 }
```

NOTES

1 A variable-string-type possesses the properties of a string-type. The individual components of a variable-string-type can be obtained by indexing it as an array (see **6.5.3.2**).

2 For additional information on the bindability and initial state of variable-string-types, see **6.4.8**.

6.4.3.4 Record-types

The structure and states of a record-type shall be the structure and states of the field-list of the record-type. The initial state denoted by a record-type shall be that denoted by 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-list-element { ';' variant-list-element }
[[';'] variant-part-completer]
| variant-part-completer) .

variant-list-element = case-constant-list ':' variant-denoter .

variant-part-completer = 'otherwise' variant-denoter .

variant-denoter = '(' field-list ')' .

variant-selector = [tag-field ':'] tag-type | discriminant-identifier .

tag-field = identifier .

tag-type = ordinal-type-name .

case-constant-list = case-range { ',' case-range } .

case-range = case-constant ['..' case-constant] .

case-constant = constant-expression .

A field-list containing neither a fixed-part nor a variant-part shall have no components, shall determine a single value-bearing state bearing a null value, shall be designated *empty*, and shall denote the totally-undefined initial state.

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 type-denoter closest-containing 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, bindability, and initial state 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 states and structure defined by the variant-part and shall have the initial state denoted by the variant-part. A variant-denoter shall not contain a type-denoter denoting either a restricted-type or the bindability that is bindable or denoting a structured-type having any component whose type-denoter is not permissible as a type-denoter contained by a variant-denoter.

Let V_i denote the state of the i-th component of a non-empty field-list having m components; then the states 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 $(F_1 * F_2 * \ldots * F_m)$.

The *variant-type* of a variant-part closest-containing either a tag-type or a discriminant-identifier in the variant-selector of the variant-part shall be the type denoted by the ordinal-type-name of the tag-type or the type possessed by the discriminant-identifier, respectively, of the variant-selector of the variant-part.

A case-constant shall denote the value denoted by the constant-expression of the case-constant. A case-range shall denote the values denoted by the case-constants of the case-range and, if two case-constants are specified, the values, if any, between the values denoted by the case-constants. If present, the second case-constant of the case-range shall denote a value greater than or equal to the value denoted by the first case-constant of the case-range and shall have the same type as the type of the first case-constant of the case-range.

The type of each case-constant of a case-range of the case-constant-list of a variant-list-element of a variant-part shall be compatible with the variant-type of the variant-part, and the value denoted by each such case-constant shall be a member of the set of values determined by that type; no value shall be denoted by more than one case-range closest-contained by the variant-part.

Each variant-denoter closest-contained by a variant-part shall denote a distinct component of the variant-part; the component shall have the structure, states, and initial state of the field-list of the variant-denoter and shall be designated a *variant* of the variant-part.

Each value denoted by a case-range of the case-constant-list of a variant-list-element shall be designated as *corresponding* to the variant denoted by the variant-denoter of the variant-list-element. Each value, if any, of the variant-type of a variant-part that is not denoted by a case-range of the case-constant-list of a variant-list-element of that variant-part shall be designated as *corresponding* to the variant denoted by the variant-denoter of the variant-part-completer of the variant-part. Each value possessed by the variant-type of a variant-part shall correspond to one and only one variant of the variant-part.

With each variant-part shall be associated a type designated the *selector-type* possessed by the variant-part. If the variant-selector of a variant-part contains a tag-field or discriminant-identifier, then the selector-type possessed by the variant-part shall be the variant-type, and each variant of the variant-part shall be *associated* with exactly those values designated as corresponding to 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 type-denoter closest-containing 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. The selector shall be nonbindable.

The initial state possessed by the selector of a variant-part type shall be determined as follows.

a) If a discriminant-identifier occurs in the variant-selector of the variant-part, the initial state shall be the state bearing the value denoted by the discriminant-identifier;

b) If the selector is a field, the initial state shall be the initial state denoted by the tag-type of the variant-selector of the variant-part;

c) If the selector is not a field and the tag-type denotes an initial state that is not undefined, the initial state shall be the state bearing a value of the selector-type; this value shall be the value associated with the variant corresponding to the value borne by the initial state denoted by the tag-type;

d) Otherwise, the initial state shall be totally-undefined.

The value of the selector of the variant-part shall cause the associated variant of the variant-part to be designated *active*. In a record-type derived from a schema with a tuple, the value of the selector of a variant-part closest-containing a variant-selector containing a discriminant-identifier shall be that value of the value corresponding to the discriminant-identifier according to the tuple; it shall be a dynamic-violation to attribute another value to such a selector (see **6.5.3.3**).

The set of states determined by a variant-part shall contain, in addition to the totally-undefined state (see **6.2.4**), the states that are the distinct pairs

$$(\mathsf{k}, \mathsf{X}_k)$$

where k represents a value of the selector-type of the variant-part and X_k is a state of the field-list of the active variant of the variant-part. The value-bearing states shall be those pairs where X_k is a value-bearing state.

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 $(T_1 + T_2 + \dots + T_n)$. 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**, **6.7.3.3**, and **6.7.5.3**.

The bindability of each field of a required record-type shall be nonbindable. If the variant-selector of the variant-part closest-contains an ordinal-type-name, the ordinal-type-name of the tag-type of the variant-selector of the variant-part shall denote the bindability that is nonbindable.

Examples:

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

2)
```
record
   name, firstname : namestring;
   age : 0..969;      { Age of Methuselah, see Genesis 5:27 }
   case married : Boolean of
     true  : (Spousesname : namestring);
     false : ( )
end
```

3)
```
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
```

4)
```
record
   field1 : integer;
   case tag : initially_42 of
     1: (field2 : real value 0.0);
     42: (field3 : integer value 13#42);
```

```
        otherwise (field4 : Boolean value false);
    end
```

There shall be a record-type designated packed and denoted by the required type-identifier **TimeStamp**. For each of the required field-identifiers **DateValid**, **TimeValid**, **year**, **month**, **day**, **hour**, **minute**, and **second**, there shall be an associated required field of the record-type, and that field shall have a type denoted by the type-denoter Boolean, Boolean, integer, 1..12, 1..31, 0..23, 0..59, and 0..59, respectively.

NOTES

4 This is analogous to the Pascal record-type:

```
packed record
   DateValid,
   TimeValid  : Boolean;
   year       : integer;
   month      : 1..12;
   day        : 1..31;
   hour       : 0..23;
   minute     : 0..59;
   second     : 0..59;
end
```

5 A processor may provide additional fields as an extension. These fields might contain information such as day of the week, fractions of seconds, leap seconds, time zone, or local time differential from Universal Time.

6 The required type-identifier **TimeStamp** is used by the time procedure **GetTimeStamp** (see **6.7.5.8**) and by the time functions **date** and **time** (see **6.7.6.9**).

There shall be a record-type designated packed and denoted by the required type-identifier **BindingType**. For each of the required field-identifiers **name** and **bound**, there shall be an associated required field of the record-type, and that field shall have an implementation-defined variable-string-type and a type denoted by the type-denoter Boolean, respectively. The values of this record-type shall designate the status of binding to external entities.

NOTES

7 A processor may provide additional fields as an extension.

8 The required type-identifier **BindingType** is used by the binding procedure **bind** (see **6.7.5.6**) and the binding function **binding** (see **6.7.6.8**).

6.4.3.5 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 **6.8.3.4**.

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 range-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 **6.8.1**).

A set-type shall denote an initial state that is totally-undefined.

An ordinal-type contained by a set-type shall denote the bindability that is nonbindable.

6.4.3.6 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, generated, or updated.

file-type = 'file' ['[' index-type ']'] 'of' component-type .

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

Examples:

```
file of real
file of vector
file [char] of 1..9999
```

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 six rules in items a) to f).

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 programming language Extended Pascal. These notations are used to define file values, below, and the required file operations elsewhere in clause **6**.

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.

f) Let x be a sequence; then the notation length(x) is 0 if x = S( ); otherwise length(x) is 1+length(x.rest).

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

NOTE — 3 The explicit denotation of the values Inspection, Generation, and Update is not part of the programming language Extended Pascal.

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).

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.

If there is an index-type in a file-type, then that file-type shall be designated a *direct-access* file-type. If f is of a direct-access file-type with index-type T, and a is the smallest value of type T and b is the largest value of type T, then it shall be an error whenever f.L and f.R are defined and length(f.L~f.R) > ord(b)-ord(a)+1.

If the file-type is not a direct-access file-type, then f.M shall not be Update.

There shall be a file-type that is not a direct-access file-type, and that type shall be denoted by the required 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 **6.7.6.5** and **6.10**.

A file-type shall denote an initial state that is totally-undefined.

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-value shall identify a distinct variable possessing a type, bindability, and initial state specified by the domain-type of the new-pointer-type that denotes the pointer-type. The domain-type shall either specify the type, bindability, and initial state denoted by the type-name of the domain-type, or specify each type, bindability, and initial state produced from the schema denoted by the schema-name of the domain-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 **6.7.5.3**).

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-name .

new-pointer-type = '↑' domain-type .

domain-type = type-name | schema-name .

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.

A new-pointer-type shall denote an initial state that is totally-undefined.

The bindability denoted by a new-pointer-type shall be nonbindable.

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 and T2 are ordinal-types and have the same range-type (see **6.4.2.1**).

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 is either a string-type (see **6.4.3.3**) or the char-type and T2 is either a string-type or the char-type.

6.4.6 Assignment-compatibility

A value of type T2 shall be designated *assignment-compatible* with a type T1 if any of the following six 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 **6.4.3.6**).

   NOTE — Because T1 and T2 are types, rather than type-denoters, the restriction on the bindability of component-types of file-types does not apply here.

b) T1 is the real-type and T2 is the integer-type.

c) T1 is the complex-type and T2 is either the integer-type or the real-type.

d) T1 and T2 are compatible ordinal-types, and the value of type T2 is in the closed interval specified by the type T1.

e) 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.

f) T1 and T2 are compatible, T1 is a string-type or the char-type, and the length of the value of T2 is less than or equal to the capacity of T1 (see **6.4.3.3**).

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 a member of the value of type T2 is not in the closed interval specified by the base-type of the type T1;

c) it shall be an error if T1 and T2 are compatible, T1 is a string-type or the char-type, and the length of the value of T2 is greater than the capacity of T1;

d) it shall be a dynamic-violation if T1 and T2 are produced from the same schema, but not with the same tuple (see **6.4.7**).

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 (see **6.4.2.2 b)**).

At any place where the rule of assignment-compatibility is used to require a value of integer-type or real-type to be assignment-compatible with a complex-type, an implicit integer-to-complex conversion or real-to-complex conversion, respectively, shall be performed (see **6.4.2.2 e)**).

At any place where the rule of assignment-compatibility is used to require a value of the char-type to be assignment-compatible with a string-type, the char-type value shall be treated as a value of the canonical-string-type with length 1 and with the component-value equal to the char-type value.

At any place where the rule of assignment-compatibility is used to require a value of the canonical-string-type to be assignment-compatible with a fixed-string-type or the char-type, the canonical-string-type value shall be treated as a value of the fixed-string-type whose components in order of increasing index shall be the components of the canonical-string-type value in order of increasing index followed by zero or more spaces.

6.4.7 Schema-definitions

A schema shall be a one-to-one mapping from a domain consisting of discriminant tuples to a set of types. Within an activation, a schema-definition containing a formal-discriminant-part shall define a new schema that is distinct both from the schema defined by the schema-definition within any other activation and from any schema defined by any other schema-definition.

schema-definition = identifier '=' schema-name
| identifier formal-discriminant-part '=' type-denoter .

formal-discriminant-part = '(' discriminant-specification { ';' discriminant-specification } ')' .

discriminant-specification = identifier-list ':' ordinal-type-name .

discriminant-identifier = identifier .

schema-identifier = identifier .

schema-name = [imported-interface-identifier '.'] schema-identifier .

A schema-name shall denote the schema denoted by the schema-identifier of the schema-name.

The occurrence of an imported-interface-identifier in a schema-name shall be the defining-point of each imported schema-identifier associated with the imported-interface-identifier for the region that is the schema-identifier of the schema-name.

NOTE — 1 'Extra' formal discriminants that do not occur in the type-denoter of the schema-definition can be used to create several distinct, but structurally-identical, types.

The occurrence of an identifier in a schema-definition of a type-definition-part of a block, a module-heading, or a module-block shall constitute its defining-point for the region that is the block, the module-heading, or the module-block, respectively. Each applied occurrence of that identifier shall be a schema-identifier. Within an activation of the block, the module-heading, or the module-block, all applied occurrences of that identifier shall denote either the schema denoted by the schema-name of the schema-definition or the new schema contained by the activation and corresponding to the schema-definition (see **6.2.3.2**). Each schema contained by an activation and corresponding to a schema-definition shall be distinct from any schema contained by any other activation and from any schema corresponding to any other schema-definition. Except for applied occurrences in the domain-type of a new-pointer-type, the schema-definition shall not contain an applied occurrence of that identifier.

The occurrence of an identifier in the identifier-list of a discriminant-specification of a formal-discriminant-part of a schema-definition shall constitute its defining-point as a discriminant-identifier for the region that is the formal-discriminant-part of the schema-definition and for the region that is the type-denoter of the schema-definition; the discriminant-identifier shall possess the type denoted by the ordinal-type-name of the discriminant-specification. Each such discriminant-identifier shall be a formal discriminant of the schema defined by the schema-definition.

The type-denoter of a schema-definition shall contain a new-type.

A formal-discriminant-part that contains the defining-points for n discriminant-identifiers, say I_1, I_2,..., I_n, in order of occurrence of their defining-points, shall determine a set of *allowed* discriminant tuples of the form

$$(V_1, V_2,..., V_n)$$

is a value belonging to the set of values determined by the type possessed by I_i. V_i and I_i shall be said to *correspond to* each other *according to* the tuple. Two such tuples shall be designated the *same* tuple if and only if they consist of the same number of values and they have equal values in corresponding positions.

Within an activation, the domain of a schema contained by the activation and corresponding to a schema-definition (see **6.2.3.2**) shall be the maximal subset of the set of tuples allowed by the formal-discriminant-part of the schema-definition, such that the schema shall associate with each tuple in its domain the type, bindability, and initial state denoted by the type-denoter of the schema-definition, with each discriminant-identifier contained by the type-denoter denoting the value corresponding to the discriminant-identifier according to the tuple. It shall be an error if the domain is empty.

NOTE — 2 A tuple allowed by the formal-discriminant-part is not in the domain of the schema if, after substitution of the tuple's constituent values for their corresponding discriminant-identifiers, one or more of the following is true within the schema-definition:

a) the first subrange-bound of a subrange-type is greater than the second subrange-bound (see **6.4.2.4**).

b) a value denoted by a discriminant-value is outside the range of the corresponding formal discriminant (see **6.4.8**).

c) a case-constant-list within an array-value in an initial-state-specifier specifies an index value that is outside the range of the corresponding index-type (see **6.8.7.2**).

d) a value denoted by an actual-discriminant-value contained by the schema-definition and corresponding to a discriminant-identifier closest-contained by a variant-selector does not correspond to one and only one variant of the variant-part.

Example:

```
type
   subrange(l,u:integer) = l..u;
   a_subrange = subrange(expression1, expression2);
   variant_record(d : a_subrange) =
       record case d of
         1: (f1 : integer);
         2: (f2 : integer);
       end;
```

The type to which a schema maps a tuple shall be said to be *produced* from the schema *with* the tuple.

An expression contained by a schema-definition shall be nonvarying (see **6.8.2**).

The ordinal-type-name of a discriminant-specification shall denote the bindability that is nonbindable.

6.4.8 Discriminated-schemata

A type denoted by a discriminated-schema shall be produced from the schema denoted by the schema-name of the discriminated-schema with the tuple denoted by the actual-discriminant-part of the discriminated-schema. The bindability denoted by the discriminated-schema shall be the bindability associated with the tuple by the schema. The initial state denoted by the discriminated-schema shall be the initial state associated with the tuple by the schema. The tuple shall consist of the values of the discriminant-values of the actual-discriminant-part taken in textual order; the type of each such discriminant-value shall be compatible with the type of the corresponding formal discriminant of the schema. It shall be a dynamic-violation if the tuple is not in the domain of the schema. A type produced from a schema with a tuple shall be distinct from a type produced from the schema with a distinct tuple and from all types produced from a distinct schema with a tuple.

discriminated-schema = schema-name actual-discriminant-part .

actual-discriminant-part = '(' discriminant-value { ',' discriminant-value } ')' .

discriminant-value = expression .

An evaluation of an actual-discriminant-part shall constitute the evaluation in implementation-dependent order of the discriminant-values in the actual-discriminant-part. Within the commencement of either an activation of a block, a module-heading, or a module-block, closest-containing a discriminant-value, the discriminant-value shall denote the value denoted by the expression in the discriminant-value. Evaluation of a discriminant-value shall constitute evaluation of the expression in the discriminant-value.

A discriminated-schema that denotes a type produced from the required schema **string** shall denote an initial state that is totally-undefined and the bindability that is nonbindable.

6.4.9 Type-inquiry

A type-inquiry shall denote a type, bindability, and initial state.

type-inquiry = 'type' 'of' type-inquiry-object .

type-inquiry-object = variable-name | parameter-identifier .

The type denoted by a type-inquiry shall be the type possessed by the variable-identifier or parameter-identifier contained by the type-inquiry. The bindability denoted by a type-inquiry shall be the bindability possessed by the variable-identifier or parameter-identifier contained by the type-inquiry. The initial state denoted by a type-inquiry shall be the initial state possessed by the variable-identifier or parameter-identifier contained by the type-inquiry. A parameter-identifier in a type-inquiry-object shall have its defining-point in a value-parameter-specification or variable-parameter-specification in the formal-parameter-list closest-containing the type-inquiry-object.

Example:

```
procedure p(var a : VVector);
var b : type of a;
    {parameter a and variable b will have the same type}
```

6.4.10 Example of a type-definition-part

```
type
   natural = 0..maxint;
   count = integer value 1;
   range = integer;
   year = 1900..1999;

{ Count, range, and integer denote the same type; range and
  integer have the same initial state (undefined). The types denoted
  by year and natural are compatible with, but not the same as,
  the type denoted by range, count, and integer.                  }

   colour = (red, yellow, green, blue);
   sex = (male, female);
   shape = (triangle, rectangle, circle);
   punchedcard = array [1..80] of char;
   charsequence = file of char;
   angle = real value 0.0;
   subpolar = record
                r : real;
                theta : angle
              end;
   indextype = 1..limit;
   vector = array [indextype] of real;
   person = ↑ persondetails value nil;
```

```
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;
initially_42 = integer value 42;
quiver = array [1..10] of vector;
sieve = set of 1..20;
FileOfInteger = file of integer;
VectorIndex = 1 .. maxint;
Bindable_FOI = bindable FileOfInteger;
VVector(vlength: VectorIndex) =
        array [1 .. vlength] of real;
Pixel = set of colour;
DeviceStatusType = (Busy, LineBreak, OutOfPaper, ParityError);
namestring = string(20);
SWidth = 0 .. 1023;
SHeight = 0 .. 2047;
Screen(width: SWidth; height: SHeight) =
         array[0 .. height, 0 .. width] of Pixel;
Positive   = 1..MaxMatrix;
Matrix(M,N : Positive) = array[1..M, 1..N] of real;
M = Matrix(5,10);
colour_map(formal_discriminant : colour) =
      record
      case formal_discriminant of
         red:    (red_field : integer value ord(red));
         yellow: (yellow_field : integer value ord(yellow));
         green:  (green_field : integer value ord(green));
         blue:   (blue_field : integer value ord(blue));
      end;
```

6.5 Declarations and denotations of variables

6.5.1 Variable-declarations

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

variable-identifier = identifier .

The occurrence of an identifier in the identifier-list of a variable-declaration of the variable-declaration-part of a block, a module-heading, or a module-block shall constitute its defining-point for the region that is the block, the module-heading, or the module-block, respectively. Each applied occurrence of that identifier shall be a variable-identifier. Within an activation of the block, the module-heading, or the module-block, all applied occurrences of that identifier shall denote the same corresponding variable (see **6.2.3.2 g)**) and shall possess the type and initial state denoted by the type-denoter of the variable-declaration. The variable-identifier shall possess the bindability denoted by the type-denoter, unless the variable-identifier is a program-parameter or a module-parameter, in which case the variable-identifier shall possess the bindability that is bindable. If the variable-identifier is a program-parameter or a module-parameter, any corresponding variable shall be designated a program-parameter or a module-parameter, respectively. The type-denoter shall not contain an applied occurrence of the identifier.

The structure of a variable possessing a structured-type shall be the structure of the structured-type.

variable-name = [imported-interface-identifier '.'] variable-identifier .

A variable-name shall denote the variable denoted by the variable-identifier of the variable-name.

The occurrence of an imported-interface-identifier in a variable-name shall be the defining-point of each imported variable-identifier associated with the imported-interface-identifier for the region that is the variable-identifier of the variable-name.

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, a buffer-variable, a substring-variable, or a function-identified-variable shall denote a declared variable, a component of a variable, a variable that is identified by an identifying-value (see **6.4.4**), a buffer-variable, a substring-variable, or a function-identified-variable (see **6.8.6.4**), respectively.

```
variable-access = entire-variable     | component-variable
                | identified-variable | buffer-variable
                | substring-variable  | function-identified-variable .
```

No statement shall threaten (see **6.9.4**) a variable-access closest-containing a protected variable-identifier (see **6.7.3.1**, **6.7.3.7.1**, and **6.11.3**).

A variable possessing the bindability that is bindable shall be totally-undefined while the variable is not bound to an external entity. It shall be an error to attribute a value to such a variable while the variable is not bound to an external entity. A variable possessing the bindability that is bindable shall possess the initial state that is totally-undefined.

The initial state of a variant of a variable possessing a variant-part type (see **6.4.3.4**) shall be:

a) if the initial state of the selector of the variable bears a value associated with the variant, the initial state possessed by the field-list of the variant-denoter that denotes the variant;

b) otherwise, totally-undefined.

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 bindable and, as a result of the binding, the execution cannot be completed as defined.

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, divvy);
  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;
  coordinate : subpolar value origin;
  pooltape : array [1..4] of FileOfInteger;
  Good_thru: record
               month : 1..12;
               year : 0..99
             end;
  MyVector : VVector(57);
  ShowScreen : Screen(759, 1023);
  DeviceStatus : set of DeviceStatusType;
  status : DeviceStatusType;
  measure : complex value polar(exp(1.0), pi);
  first_name, last_name, full_name : namestring;
  middle_initial : char;
```

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

6.5.2 Entire-variables

entire-variable = variable-name .

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 state of the component of a variable shall be the same component of the state of the variable. The components of a variable possessing a string-type shall have the bindability that is nonbindable. It shall be an error to access or reference a component of a variable that possesses the bindability that is bindable while the variable is not bound to an external entity.

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

6.5.3.2 Indexed-variables

An indexed-variable shall denote a component of a variable possessing an array-type or a string-type.

indexed-variable = array-variable '[' index-expression { ',' index-expression } ']'
| string-variable '[' index-expression ']' .

array-variable = variable-access .

string-variable = variable-access .

index-expression = expression .

An array-variable shall be a variable-access that denotes a variable possessing an array-type. A string-variable shall be a variable-access that denotes a variable possessing a string-type. The string-variable of an indexed-variable shall denote a variable possessing a variable-string-type.

NOTE — 1 Variables possessing a fixed-string-type are indexed using array-type properties.

For an array-variable in 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 of the array-variable.

For a string-variable in an indexed-variable, the index-expression of the indexed-variable shall possess the integer-type, and it shall be an error if the value of the index-expression is not in the index-domain of the value of the string-variable. It shall be an error to alter the length of the value of a string-variable when a reference to a component of the string-variable exists. It shall be an error to access an indexed-variable when the string-variable, if any, of the indexed-variable is undefined.

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**) or string-variable (see **6.4.3.3**).

Examples:

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

If the array-variable or string-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 or string-variable of, an indexed-variable shall be implementation-dependent.

Examples:

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

NOTE — 2 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 **6.4.3.4**) with the field-identifier of the field-specifier of the field-designator or shall denote the variable denoted by the field-designator-identifier (see **6.9.3.10**) 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
Good_thru.year
```

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 the value associated (see **6.4.3.4**) with the variant.

It shall be an error unless a variant of a record-variable 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.

NOTES

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

2 When a variant *becomes active*, it is not *created* and therefore its initial state does not apply.

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 **6.7.5.3**).

identified-variable = pointer-variable '↑' .

pointer-variable = variable-access .

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 **6.7.5.3**) 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. The initial state possessed by a buffer-variable shall be totally-undefined. The bindability possessed by a buffer-variable shall be nonbindable.

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.

6.5.6 Substring-variables

A substring-variable shall denote a variable possessing a new fixed-string-type. The bindability possessed by the substring-variable shall be nonbindable.

substring-variable = string-variable '[' index-expression '..' index-expression ']' .

The index-expressions in a substring-variable shall possess the integer-type. It shall be an error if the string-variable of the substring-variable is undefined, or if the value of an index-expression in a substring-variable is less than 1 or greater than the length of the value of the string-variable of the substring-variable, or if the value of the first index-expression is greater than the value of the second index-expression. The capacity of the fixed-string-type possessed by the variable denoted by the substring-variable shall be equal to one plus the value of the second index-expression minus the value of the first index-expression. The components of the variable denoted by the substring-variable shall be, in order of increasing index, the contiguous components of the string-variable from the component that corresponds to the value of the first index-expression through the component that corresponds to the value of the second index-expression.

The order of both the evaluation of the index-expressions of, and the access to the string-variable of, a substring-variable shall be implementation-dependent.

It shall be an error to alter the length of the value of a string-variable when a reference to a substring of the string-variable exists. A reference or an access to a substring of a variable shall constitute a reference or access, respectively, to the variable.

6.6 Initial states

The initial state specified by an initial-state-specifier shall be the state bearing the value denoted by the component-value of the initial-state-specifier.

initial-state-specifier = 'value' component-value .

An expression contained by the component-value of an initial-state-specifier shall be nonvarying (see **6.8.2**). The type of a component-value of an initial-state-specifier of a type-denoter shall be the type denoted by the type-denoter.

NOTES

1 Within an activation, the component-value of an initial-state-specifier denotes one or more constant values.

2 Each state of a variable that has no value attributed to it is undefined. The state of a variable that has no value attributed to it, and whose components are totally-undefined, is totally-undefined. See **6.2.4**.

3 When a type-denoter closest-contains a new-structured-type, the initial-state-specifier is associated with (and therefore must be compatible with) the entire structured-type, not with its component-type or base-type (as appropriate). For example

```
type  S = array [1..8] of char value [1..8: '*'];
```

is valid, and the initial state denoted by S is an array of eight stars; whereas

```
type  S = array [1..8] of char value '*';
```

is a violation.

4 The component-value of an initial-state-specifier consists of an assignment-compatible expression, an array-value, or a record-value (see **6.8.7.1**).

6.7 Procedure and function declarations

6.7.1 Procedure-declarations

procedure-declaration = procedure-heading ';' remote-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 .

procedure-name = [imported-interface-identifier '.'] procedure-identifier .

A procedure-name shall denote the procedure denoted by the procedure-identifier of the procedure-name.

The occurrence of an imported-interface-identifier in a procedure-name shall be the defining-point of each imported procedure-identifier associated with the imported-interface-identifier for the region that is the procedure-identifier of the procedure-name.

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 or module-block closest-containing the procedure-declaration. The occurrence of an identifier in a procedure-heading of a procedure-and-function-heading-part contained by a module-heading shall constitute its defining-point as a procedure-identifier for the region that is the module-heading. Within an activation of that block, that module-heading, or that module-block, each applied occurrence of the identifier shall denote the corresponding procedure (see **6.2.3.2**).

Each identifier having a defining-point as a procedure-identifier in a procedure-heading of a procedure-declaration in which the remote-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.

Each identifier having a defining-point as a procedure-identifier in a procedure-heading of a procedure-and-function-heading-part of a module-heading shall have exactly one of its applied occurrences in a procedure-identification of a procedure-declaration of a procedure-and-function-declaration-part of the module-block that is associated with the module-heading (see **6.11.1**).

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.

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.

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 = 10.0 * epsreal;
  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.  Thus it has
  the same effect as the required procedure extend (see 6.7.5.2).
  Simpler conditioning is possible (without using extend) only if
  additional 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 };

procedure SumVectors (var A, B, C : VVector);
var
  i : VectorIndex;
begin
  for i := 1 to A.vlength do A[i] := B[i] + C[i];
end   { of SumVectors };
```

6.7.2 Function-declarations

function-declaration = function-heading ';' remote-directive
| function-identification ';' function-block
| function-heading ';' function-block .

function-heading = 'function' identifier [formal-parameter-list]
[result-variable-specification] ':' result-type .

result-variable-specification = '=' identifier .

function-identification = 'function' function-identifier .

function-identifier = identifier .

result-type = type-name .

function-block = block .

function-name = [imported-interface-identifier '.'] function-identifier .

A function-name shall denote the function denoted by the function-identifier of the function-name.

The occurrence of an imported-interface-identifier in a function-name shall be the defining-point of each imported function-identifier associated with the imported-interface-identifier for the region that is the function-identifier of the function-name.

The occurrence of the identifier in a result-variable-specification of a function-heading shall constitute its defining-point as a function-result-identifier for the region that is the formal-parameter-list, if any, of the function-heading and shall constitute its defining-point as a variable-identifier for the region that is the block of the function-block, if any, associated with the identifier of the function-heading; the variable-identifier shall possess the type, initial state, and bindability denoted by the type-name of the result-type of the function-heading, and within each activation of the function-block, if any, shall denote the result of the activation (see **6.2.3.2 k)**).

If there is a result-variable-specification in the function-heading associated with a function-block, the function-block shall contain no assignment-statement (see **6.9.2.2**) such that the function-identifier of the assignment-statement is associated with the function-block, and the function-block shall contain at least one statement threatening (see **6.9.4**) a variable-access denoting the result of each activation (see **6.2.3.2 k)**) of the function-block; otherwise, the function-block shall contain at least one assignment-statement such that the function-identifier of the assignment-statement is associated with the function-block.

The occurrence of an identifier in the function-heading of a function-declaration shall constitute its defining-point as a function-identifier for the region that is the block or module-block closest-containing the function-declaration. The type and initial state associated with the function-identifier shall be the type and initial state denoted by the result-type of the function-heading. The occurrence of an identifier in the function-heading of a procedure-and-function-heading-part contained by a module-heading shall constitute its defining-point as a function-identifier for the region that is the module-heading. Within an activation of that block, that module-heading, or that module-block, each applied occurrence of the identifier shall denote the corresponding function (see **6.2.3.2**).

A type-name shall not be permissible as the type-name of a result-type if it denotes a file-type, a structured-type having any component whose type-denoter is not permissible as a component-type of a file-type, or the bindability that is bindable.

Each identifier having a defining-point as a function-identifier in the function-heading of a function-declaration in which the remote-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.

NOTE — This prohibits using a forward-declared function in a discriminated-schema and then using the type defined by that discriminated-schema inside the block of the function.

Each identifier having a defining-point as a function-identifier in a function-heading of a procedure-and-function-heading-part of a module-heading shall have exactly one of its applied occurrences in a function-identification of a function-declaration of a procedure-and-function-declaration-part of the module-block that is associated with the module-heading (see **6.11.1**).

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 type and initial state that is associated with the identifier or function-identifier. There shall be at most one function-block associated with a function-identifier.

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.

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.}
const
  eps = 10.0 * epsreal;
var
  old, estimate : real;
begin
 estimate := x;
 repeat
   old := estimate;
   estimate := (old + x / old) * 0.5;
 until abs(estimate - old) < eps * estimate;
 Sqrt := estimate
end { of Sqrt };

function max (a : vector) = largestsofar : real;
```

```
{This function finds the largest component of the value of a.}
var
  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])}
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 by examples in this 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;

function start_of_day_for(protected targ_time : TimeStamp)
        = midnight : TimeStamp;

begin
midnight := targ_time;
with midnight do begin
   hours := 0;
   minutes := 0;
   seconds := 0;
   end;
end;
```

6.7.3 Parameters

6.7.3.1 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 **6.7.3.7.1**.

value-parameter-specification = ['protected'] identifier-list ':' parameter-form .

variable-parameter-specification = ['protected'] 'var' identifier-list ':' parameter-form .

parameter-form = type-name | schema-name | type-inquiry .

parameter-identifier = 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. If either the value-parameter-specification or the variable-parameter-specification contains protected, then every type possessed by the associated variable-identifier shall be protectable, and the associated variable-identifier shall be designated *protected* (see **6.5.1**). The parameter-form of the value-parameter-specification or variable-parameter-specification shall not contain an applied occurrence of the parameter-identifier.

NOTE — 2 The state (or value, if any) of a protected formal variable parameter can change during an activation due to changes made to the actual-parameter (e.g., aliasing), whereas the value of a protected formal value parameter cannot change.

Example:

```
procedure illustrate(a : integer;   { value param }
                 var b : integer;   { variable param }
       protected     c : integer;   { protected value param }
       protected var d : integer);  { protected variable param }


{ Note: The presence of ‘protected’ on a value parameter is not        }
{    redundant as it may seem.  It indicates to the reader and the     }
{    processor that the value cannot change within the procedure.      }

begin
a := 1; { modifies local copy of parameter }
b := 1; { modifies actual variable }
{   c := 1;    not legal }
{   d := 1;    not legal }
end;
```

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 — 3 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.

6.7.3.2 Value parameters

An actual-parameter contained in the activation-point of an activation of a block and corresponding to a formal value parameter of the block shall be an expression. Within the activation, the formal-parameter and its associated variable-identifier shall denote the variable contained by the activation and corresponding to the variable-identifier (see **6.2.3.2**). Within the commencement (see **6.2.3.8**) of the activation, the value of the expression shall be attributed to the variable. The type possessed by the formal-parameter shall be one permitted as the component-type of a file-type (see **6.4.3.6**).

If the parameter-form of the value-parameter-specification contains a schema-name that denotes the schema denoted by the required schema-identifier **string**, then each corresponding actual-parameter contained by the activation-point of an activation shall possess a type having an underlying-type that is a string-type or the char-type; it shall be an error if the values of these underlying-types, associated with the values denoted by the actual-parameters, do not all have the same length. Within the activation, each corresponding formal-parameter shall possess the type produced from the schema **string** with the tuple having that length as its component. The initial state of the formal-parameters shall be totally-undefined. The formal-parameters and their associated variable-identifiers shall possess the bindability that is nonbindable.

If the parameter-form of the value-parameter-specification contains a schema-name that does not denote the schema denoted by the required schema-identifier **string**, each corresponding actual-parameter contained by the activation-point of an activation shall possess a type having the underlying-type produced from the schema denoted by the schema-name with a tuple, and it shall be a dynamic-violation if these tuples are not the same. Within the activation, the corresponding formal-parameter shall possess the type produced from the schema with that tuple. The initial state of the formal-parameters shall be the initial state associated with the tuple by the schema. The bindability of the formal-parameters and their associated variable-identifiers, and the bindability associated by the schema with each tuple in the schema's domain, shall be nonbindable.

NOTE — If the types derived from such a schema are subrange-types or set-types then no actual-parameter expression can satisfy these requirements since a primary of a subrange-type is treated as if it were of the host-type and a primary of a set-type is treated as if it were of the appropriate unpacked-canonical-set-of-T-type or packed-canonical-set-of-T-type.

If the parameter-form of the value-parameter-specification contains a type-name or a type-inquiry, each formal-parameter associated with an identifier in the identifier-list in that value-parameter-specification shall possess the type denoted by the type-name or type-inquiry, respectively. The value in the underlying-type of the type of each corresponding actual-parameter, associated with the value of the actual-parameter (see **6.4.2.5**), shall be assignment-compatible with the type possessed by the formal-parameters. The initial state of the formal-parameters shall be the initial state denoted by the type-name or type-inquiry. The bindability of the formal-parameters and their associated variable-identifiers, and the bindability denoted by the type-name or type-inquiry, shall be nonbindable.

6.7.3.3 Variable parameters

An actual-parameter contained in the activation-point of an activation of a block and corresponding to a formal variable parameter of the block shall be a variable-access. Within the commencement (see **6.2.3.8**) of the activation, the actual-parameter shall be accessed, and this access shall establish a reference, contained by the activation (see **6.2.3.2**), to the accessed variable. Within the activation, the variable-identifier associated with the formal-parameter shall denote the corresponding referenced variable. The formal-parameter and its associated variable-identifier shall possess the bindability that is possessed by the actual-parameter.

NOTE — 1 The actual-parameter may possess any bindability if it possesses a file-type, in which case the bindability of the formal-parameter is determined dynamically by the actual-parameter.

If the parameter-form of the variable-parameter-specification contains a schema-name, all of the corresponding actual-parameters contained by the activation-point of an activation shall possess the same underlying-type (see **6.4.2.5**) that is produced from the schema denoted by the schema-name with a tuple. Within the activation, each corresponding formal-parameter shall possess that type. The initial state of the formal-parameter shall be the initial state associated with the tuple by the schema. The formal-parameters and their associated variable-identifiers shall possess the bindability associated by the schema with each tuple in the schema's domain, unless a type produced from the schema with such a tuple is a file-type.

NOTE — 2 For an example of a procedure with a schema variable parameter, see program 4) in **6.12**.

A type produced from a schema with a tuple shall be designated *schematic*. A type denoted by a restricted-type shall be designated *schematic* if the underlying-type of the restricted-type is schematic. Either both of the types of a formal-parameter and its corresponding actual-parameter shall be schematic, or neither of the types shall be schematic.

If the parameter-form of a variable-parameter-specification contains a type-name or a type-inquiry and the underlying-type of the type denoted by the type-name or type-inquiry is not schematic, either the types possessed by the formal-parameter and the actual-parameter shall be the same type or the type possessed by one shall be the underlying-type of the type possessed by the other. If the parameter-form of a variable-parameter-specification contains a type-name or a type-inquiry and the underlying-type of the type denoted by the type-name or type-inquiry is schematic, either the types possessed by the formal-parameter and the actual-parameter shall be produced from the same schema or the type possessed by one shall be produced from the same schema as the underlying-type of the type possessed by the other, and it shall be a dynamic-violation if underlying-types of the types possessed by the formal-parameter and actual-parameter are produced from the same schema, but not with the same tuple. The initial state of the formal-parameters shall be the initial state denoted by the type-name or type-inquiry. The formal-parameters and their associated variable-identifiers shall possess the bindability denoted by the type-name or type-inquiry, unless the type-name or type-inquiry denotes a file-type.

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. An actual variable parameter shall not denote a component of a string-type.

NOTE — 3 An actual variable parameter cannot denote a substring-variable because the type of a substring-variable is a new fixed-string-type different from every named type.

6.7.3.4 Procedural parameters

An actual-parameter contained in the activation-point of an activation of a block and corresponding to a formal procedural parameter of the block shall be a procedure-name. Within the activation, the formal-parameter and its associated procedure-identifier shall denote the procedure denoted by the actual-parameter. The procedure shall be one that is contained by an activation. 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.

6.7.3.5 Functional parameters

An actual-parameter contained in the activation-point of an activation of a block and corresponding to a formal functional parameter of the block shall be a function-name. Within the activation, the formal-parameter and its associated function-identifier shall denote the function denoted by the actual-parameter. The function shall be one that is contained by an activation. 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. If the result-type closest-contained by the formal-parameter-section denotes a type not produced from a schema, that result-type shall denote the same type as the type of the function; otherwise, the type denoted by the result-type shall be produced from the same schema as the type of the function, and it shall be a dynamic-violation if the type denoted by the result-type and the type of the function are produced from the same schema, but not with the same tuple.

NOTES

1 Since required procedures and functions are not contained by an activation, they may not be used as actual-parameters.

2 For examples of the use of procedural parameters and functional parameters, see examples 6 through 9 in **6.11.6** and example 3 in **6.12**.

6.7.3.6 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 all of the statements in at least one of the following sections are true.

a) 1) They are both value-parameter-specifications containing the same number of parameters.

2) Either both contain protected or neither contains protected.

3) All the parameters possess the same bindability.

4) The schema-name in the parameter-form of each value-parameter-specification denotes the same schema, or the type-name in the parameter-form of each value-parameter-specification denotes the same type not produced from a schema, or the type-name in the parameter-form of each value-parameter-specification denotes a type that is produced from the same schema, or the variable-name closest-contained by the type-inquiry in the parameter-form of each value-parameter-specification denotes the same variable, or the parameter-identifier closest-contained by the type-inquiry in the parameter-form of each value-parameter-specification denotes parameter-identifiers with their defining-points in corresponding positions in the formal-parameter-list closest-contained by the formal-parameter-section and the formal-parameter-list that defines the formal-parameters of the procedure or function denoted by the actual-parameter. It shall be a dynamic-violation if the type-name in the parameter-form of each value-parameter-specification denotes a type produced from the same schema but not with the same tuple.

b) 1) They are both variable-parameter-specifications containing the same number of parameters.

2) Either both contain protected or neither contains protected.

3) Unless the parameters possess a file-type, all the parameters possess the same bindability.

4) The schema-name in the parameter-form of each variable-parameter-specification denotes the same schema, or the type-name in the parameter-form of each variable-parameter-specification denotes the same type not produced from a schema, or the type-name in the parameter-form of each variable-parameter-specification denotes a type that is produced from the same schema, or the variable-name closest-contained by the type-inquiry in the parameter-form of each variable-parameter-specification denotes the same variable, or the parameter-identifier closest-contained by the type-inquiry in the parameter-form of each variable-parameter-specification denotes parameter-identifiers with their defining-points in corresponding positions in the formal-parameter-list closest-contained by the formal-parameter-section and the formal-parameter-list that defines the formal-parameters of the procedure or function denoted by the actual-parameter. It shall be a dynamic-violation if the type-name in the parameter-form of each variable-parameter-specification denotes a type produced from the same schema but not with the same tuple.

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-names 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-forms. Two conformant-array-forms shall be equivalent if all of the following four statements are true and either both contain protected or neither contains protected.

1) There is a single index-type-specification in each conformant-array-form.

2) The ordinal-type-name in each index-type-specification denotes the same type.

3) Either the (component) conformant-array-forms of the conformant-array-forms are equivalent or the type-names of the conformant-array-forms denote the same type and bindability.

4) Either both conformant-array-forms are packed-conformant-array-forms or both are unpacked-conformant-array-forms.

NOTES

1 The abbreviated conformant-array-form and its corresponding full form are equivalent (see **6.7.3.7**).

2 For the status of item e) see **5.1 a)**, **5.1 b)**, **5.1 c)**, **5.2 a)**, and **5.2 b)**.

6.7.3.7 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)**.

6.7.3.7.1 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*. If the conformant-array-parameter-specification contains protected, then the variable-identifier shall be designated *protected* (see **6.5.1**).

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 =
    ['protected'](value-conformant-array-specification
        | variable-conformant-array-specification) .

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

variable-conformant-array-specification = 'var' identifier-list ':' conformant-array-form .

conformant-array-form = packed-conformant-array-form
    | unpacked-conformant-array-form .

packed-conformant-array-form = 'packed' 'array' '[' index-type-specification ']'
'of' type-name .

unpacked-conformant-array-form = 'array' '[' index-type-specification
{ ';' index-type-specification } ']'
'of' (type-name | conformant-array-form) .

index-type-specification = identifier '..' identifier ':' ordinal-type-name .

primary > bound-identifier .

bound-identifier = identifier .

NOTE — 1 There are also syntax rules for formal-parameter-section in **6.7.3.1** and for primary in **6.8.1**.

If a conformant-array-form closest-contains a conformant-array-form, 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 **6.7.3.8**) 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 constant nor a variable.

The conformant-actual-variables (see **6.7.3.7.2**) corresponding to formal-parameters that occur in a single value-conformant-array-specification and contained by one activation shall all possess the same type or shall all possess fixed-string-types with the same capacity. The conformant-actual-variables (see **6.7.3.7.3**) corresponding to formal-parameters that occur in a single variable-conformant-array-specification and having references contained by one activation shall all possess the same type. The type possessed by the conformant-actual-variables shall be conformable (see **6.7.3.8**) with the conformant-array-form, 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 conformant-actual-variables that correspond (see **6.7.3.8**) to the index-type-specifications contained by the conformant-array-form contained by the conformant-array-parameter-specification. The type and initial state denoted by the type-name that is not contained by an index-type-specification and that is contained by a conformant-array-parameter-specification shall be designated the *fixed-component-type* and *fixed-component-initial-state*, respectively, of the conformant-array-parameters defined by that conformant-array-parameter-specification. The formal-parameters shall possess the initial state of their type having as its component initial state the fixed-component-initial-state. The formal-parameters shall possess the bindability that is nonbindable.

It shall be an error if the conformant-actual-variables corresponding to formal-parameters that occur in a single value-conformant-array-specification possess fixed-string-types that have different capacities or that are not conformable with the conformant-array-form.

NOTE — 3 Although the type of an actual-parameter corresponding to a conformant-array-parameter-specification can be a string-type, the type possessed by the formal-parameter cannot be a fixed-string-type (see **6.4.3.3.2**) because the type of the formal-parameter is not denoted by the syntax of an array-type.

6.7.3.7.2 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. Within the commencement of an activation, the expression of an actual-parameter corresponding to a formal-parameter shall be evaluated, and the value thereof shall be attributed to the variable contained by the activation and corresponding to the defining-point of the variable-identifier associated with the formal-parameter (see **6.2.3.2 g)**). Within the activation, the formal-parameter and its associated variable-identifier shall denote the variable. The variable shall be designated a *conformant-actual-variable corresponding to* the formal-parameter. If the type possessed by the expression is the char-type or a string-type, then this variable shall possess a fixed-string-type with a capacity equal to the length of the value of the expression; otherwise, the type possessed by this variable shall be the same as that possessed by the expression. The value of the expression shall be assignment-compatible with the type of this variable.

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.

6.7.3.7.3 Variable conformant arrays

The identifier-list in a variable-conformant-array-specification shall be a list of variable conformant arrays. Each actual-parameter corresponding to a formal variable parameter shall be a variable-access, and each variable it denotes shall be designated a *conformant-actual-variable corresponding to* the formal-parameter. The variable denoted by the actual-parameter for an activation shall be accessed within the commencement of the activation, and the reference contained by the activation (see **6.2.3.2 h)**) shall be to the accessed variable; within the activation, the corresponding formal-parameter and its associated variable-identifier shall denote the referenced variable.

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

6.7.3.8 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-form 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-forms 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-name of the index-type-specification closest-contained by a conformant-array-form closest-containing a single index-type-specification; then T1 shall be conformable with the conformant-array-form if all the following five 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 and bindability as that denoted by the type-name of the packed-conformant-array-form or unpacked-conformant-array-form of the conformant-array-form or is conformable to the conformant-array-form closest-contained by the conformant-array-form.

d) Either T1 is not designated packed and the conformant-array-form is an unpacked-conformant-array-form, or T1 is designated packed and the conformant-array-form is a packed-conformant-array-form.

e) T1 denotes the bindability that is nonbindable.

NOTE — 2 The abbreviated and full forms of a conformant-array-form are equivalent (see **6.7.3.7**). 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.

6.7.4 Required procedures and functions

The required procedure-identifiers and function-identifiers and the corresponding required procedures and functions shall be those specified in **6.7.5**, **6.7.6**, and **6.10**.

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

6.7.5 Required procedures

6.7.5.1 General

The required procedures shall be file handling procedures, dynamic allocation procedures, transfer procedures, string procedures, binding procedures, control procedures, and time procedures.

6.7.5.2 File handling procedures

Except for the application of **rewrite**, **extend,** or **reset** to the required textfiles **input** or **output** (see **6.11.4.2**), the effects of applying each of the file handling procedures **rewrite**, **extend**, **put**, **update**, **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 effects of applying each of the file handling procedures **SeekWrite**, **SeekRead**, and **SeekUpdate** to f and n, wherein f shall be a file-variable that possesses a direct-access file-type with index-type T and n shall be an expression whose value is assignment-compatible with T, shall be defined by pre-assertions and post-assertions about f, its components f.L, f.R, and f.M, the associated buffer-variable f↑, n, and the smallest value a of type T. 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 — 1 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).

extend(f)

pre-assertion: The components f0.L and f0.R are not undefined.

post-assertion: (f.M = Generation) and (f.L = f0.L~f0.R~ X) and (f.R = S( )) and
(f↑ is totally-undefined),

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( ).

put(f)

pre-assertion: (f0.M = Generation or f0.M = Update) and (neither f0.L nor f0.R is undefined) and
(f0.R = S( ) or f is of a direct-access file-type) and
(f0↑ is not undefined).

post-assertion: (f.M = f0.M) and (f.L = f0.L~ S(f0↑)) and (if f0.R = S( )
then (f.R = S( ))
else (f.R = f0.R.rest)) and

(if (f.R = S( )) or (f0.M = Generation)
then (f↑ is totally-undefined)
else (f↑ = f.R.first)).

update(f)

pre-assertion: (f0.M = Generation or f0.M = Update) and (neither f0.L nor f0.R is undefined) and
(f is of a direct-access file-type) and (f0↑ is not undefined).

post-assertion: (f.M = f0.M) and (f.L = f0.L) and
(if f0.R = S( )
then (f.R = S(f0↑))
else (f.R = S(f0↑)~f0.R.rest)) and
(f↑ = f0↑).

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 or f0.M = Update) and (neither f0.L nor f0.R is undefined) and
(f0.R <> S( )).

post-assertion: (f.M = f0.M) 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)).

SeekWrite(f, n)

pre-assertion: (neither f0.L nor f0.R is undefined) and
(0 <= ord(n)-ord(a) <= length(f0.L~f0.R))

post-assertion: (f.M = Generation) and
(f.L~f.R = f0.L~f0.R) and
(length(f.L) = ord(n)-ord(a)) and
(f↑is totally-undefined).

SeekRead(f, n)

pre-assertion: (neither f0.L nor f0.R is undefined) and
(0 <= ord(n)-ord(a) <= length(f0.L~f0.R)).

post-assertion: (f.M = Inspection) and
(f.L~f.R = f0.L~f0.R) and
(if length(f0.L~f0.R) > ord(n)-ord(a)
then
((length(f.L) = ord(n)-ord(a)) and
(f↑= f.R.first))
else
((f.R = S( )) and
(f↑is totally-undefined))).

SeekUpdate(f, n)

pre-assertion: (neither f0.L nor f0.R is undefined) and
(0 <= ord(n)-ord(a) <= length(f0.L~f0.R))

post-assertion: (f.M = Update) and
(f.L~f.R = f0.L~f0.R) and
(if (length(f0.L~f0.R) > ord(n)-ord(a)
then
((length(f.L) = ord(n) - ord(a)) and
(f↑= f.R.first))
else
((f.R = S( )) and
(f↑ is totally-undefined))).

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 the 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 — 2 The variable-access is not a variable parameter. Consequently, it may be a variant-selector or 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,e_1); write(ff,e_2,...,e_n) 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

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

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

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

6.7.5.3 Dynamic allocation procedures

new(p)

shall create a new variable that is in its initial state and not bound to an external entity; 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 domain-type of the new-pointer-type denoting the type possessed by p shall contain a type-identifier, which shall denote the type, bindability, and initial state of the created variable.

new(p,c_l,...,c_n)

shall create a new variable that is in its initial state and not bound to an external entity; 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 domain-type of the new-pointer-type denoting the type possessed by p shall contain a type-identifier, which shall denote the record-type, the bindability, and except as otherwise noted below, the initial state of the created variable.

The case-constant c_1 shall correspond to the variant-part of the field-list of the record-type. For $1 < i <= n$, c_i shall correspond to the variant-part of the field-list of the variant-denoter denoting the variant specified by c_i-1. For $1 <= i <= n$, the variant-part corresponding to c_i shall closest-contain a tag-type. For $1 <= i <= n$, the initial state of the selector of the variant corresponding (see above) with the case-constant c_i shall be the state bearing the value associated (see **6.4.3.4**) with the variant corresponding (see **6.4.3.4**) to the value denoted by c_i.

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.

NOTE — 1 Since the initial state of each selector is determined by the corresponding case-constant, any corresponding tag-field is also attributed the value of the case-constant (see **6.4.3.4**).

new(p,d_1,...,d_s)
shall create a new variable that is in its initial state and not bound to an external entity; 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 domain-type of the new-pointer-type denoting the type possessed by p shall contain a schema-identifier. The created variable shall possess the type, bindability, and initial state associated by the schema denoted by the schema-identifier with the tuple consisting of the values of the expressions d_1,...,d_s taken in textual order; the type of each such expression shall be compatible with the type of the corresponding formal discriminant of the schema. The order of evaluation of the expressions shall be implementation-dependent.

It shall be a dynamic-violation if the tuple is not in the domain of the schema.

NOTES

2 If the schema is the required schema **string**, then $s = 1$, and the created variable possesses a new variable-string-type with capacity equal to the value of d_1, a positive integer.

3 The variable-access p is not a variable parameter. Consequently, it may be a variant-selector or a component of a packed structure.

A variable created by the required procedure **new** shall exist until the termination of the activation of the program-block or until the value identifying the variable is removed from the set of values of its pointer-type.

NOTE — 4 A complying program can access an identified-variable only when the identifying-value is attributed to a variable (possibly a function activation result).

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_l,...,c_n).

dispose(q,k_l,...,k_m)
shall remove the identifying-value denoted by the expression q from the pointer-type of q. The case-constants k_l,...,k_m shall be listed in order of increasing nesting of the variant-parts, each

closest-containing a tag-type. It shall be an error unless the variable had been created using the form new(p,c_l,...,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 — 5 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 **6.7.3.2** and **6.9.2.2**).

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

It shall be an error if a variable-access in a primary, in an assignment-statement, or in an actual-parameter closest-contains an identified-variable that denotes a variable created using the form new(p, c_l,...,c_n).

6.7.5.4 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 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.

6.7.5.5 String procedures

readstr(e,v_1,...,v_n)

The syntax of the parameter list of **readstr** shall be

readstr-parameter-list = '(' string-expression ',' variable-access { ',' variable-access } ')' .

string-expression = expression .

The expression of a string-expression shall possess char-type or canonical-string-type.

Apart from the restrictions imposed by requirements given in this clause, the execution of readstr(e,v_1,...,v_n) where e denotes a string-expression and v_1,...,v_n denote variable-accesses possessing the char-type (or a subrange of char-type), the integer-type (or a subrange of integer-type), the real-type, a fixed-string-type, or a variable-string-type, shall be equivalent to

```
begin
  rewrite(f);
  writeln(f, e);
  reset(f);
  read(f, v1,...,vn)
end
```

where f denotes an auxiliary variable that the program does not otherwise contain, which possesses the required type **text**. (See **6.10.1 b)**, **6.10.1 c)**, **6.10.1 d)**, **6.10.1 e)**, and **6.10.1 f)**.) It shall be an error if the equivalent of eof(f) is true upon completion.

NOTE — 1 The value of the string-expression must contain a representation of a value for each variable-access. It may contain other representations after these, but they are not 'read'.

Example:

```
E := '0.0-4';
readstr (E, R, C, I);
```

NOTE — 2 The above example, where E, R, C, and I possess a variable-string-type having a capacity of at least 5, the real-type, the char-type, and the integer-type, respectively, yields:

```
R = 0.0,
C = '-', and
I = 4.
```

writestr(s,p_1,...,p_n)

The syntax of the parameter list of **writestr** shall be

writestr-parameter-list = '(' string-variable ',' write-parameter { ',' write-parameter } ')' .

NOTE — 3 Write-parameter is defined in **6.10.3**.

Writestr(s,p_1,...,p_n) shall access the string-variable s, which shall possess a fixed-string-type or a variable-string-type, and establish a reference to that string-variable for the remaining execution of the statement. The execution of the statement shall be equivalent to

```
begin
  rewrite(f);
  writeln(f,pl,...,pn);
  reset(f);
  read(f,ss)
end
```

where ss denotes the referenced string-variable corresponding to s, and f denotes an auxiliary variable that the program does not otherwise contain, which possesses the required type **text**. It shall be an error if any of the write-parameters accesses the referenced string-variable. It shall be an error if the equivalent of eoln(f) is false upon completion.

NOTE — 4 The capacity of the string-type possessed by the string-variable must be great enough to receive the concatenation of the representations of the values specified by the write-parameters.

Example:

```
writestr(S, 0.168:5:2, 6:3);
```

NOTE — 5 The above example, where S possesses a string-type having a capacity of at least 8, might yield (assuming that type **real** is sufficiently precise):

```
S = ' 0.17  6'.
```

6.7.5.6 Binding procedures

bind(f,b)

For the variable-access f, and the expression b that shall possess the type denoted by the required type-identifier **BindingType** (see **6.4.3.4**), the statement bind(f,b) shall access the variable denoted by f and shall attempt to bind the accessed variable to an entity that is external to the program and that is designated by b. The binding shall be implementation-defined. It shall be a dynamic-violation if the variable is already bound to an external entity. If the variable-access f possesses a file-type, it shall be a dynamic-violation if the variable does not possess the bindability that is bindable; otherwise, the variable shall possess the bindability that is bindable.

NOTES

1 The procedure **bind** may change the state of the variable that is to be bound in an implementation-defined way.

2 The function **binding** (see **6.7.6.8**) can be used to obtain an initial value of type **BindingType** and to test the success of binding a variable to an external entity.

3 The value of b.bound is ignored by bind(f,b). In particular, b.bound is not required to be false (although it is an error if f is already bound to an external entity); and b.bound being false does not make bind(f,b) equivalent to unbind(f).

4 In bind(f,b), b may be any expression of type **BindingType**; but even if b is a variable, the value of b is not altered by bind(f,b). In particular, bind(f,b) does not set b.bound to true or false to reflect the success of the binding. The only time b.bound is guaranteed to be the binding status of f is immediately after a statement such as b:=binding(f) (see **6.7.6.8**).

5 After bind(f,b), the value of b is altered only by program action. Bind(f,b) binds f to the external entity described by b; it does not set up any dynamic association between the binding and b.

6 An example is found in **6.7.6.8**.

unbind(f)

For a variable-access f, the statement unbind(f) shall access the variable denoted by f and shall attempt to unbind the accessed variable from the entity external to the program to which it is bound, if any. If the attempt is successful, the variable shall become totally-undefined. The effect on the binding, if any, of any bindable variable contained by the accessed variable shall be implementation-dependent. If the variable-access f possesses a file-type, it shall be a dynamic-violation if the variable does not possess the bindability that is bindable; otherwise, the variable shall possess the bindability that is bindable.

NOTE — 7 Unbind(f) is permitted even if f is not bound to an external entity and is permitted even if f is totally-undefined.

6.7.5.7 Control procedures

halt

Following execution of the control procedure **halt** within an activation of a program, no further processing (see 3.6) of the activation of the program shall occur.

6.7.5.8 Time procedures

GetTimeStamp(t)

The variable-access t shall possess the type denoted by the required type-identifier **TimeStamp** (see **6.4.3.4**).

The procedure shall attribute to the variable denoted by the variable-access t either a value whose field **DateValid** represents the value true and whose fields **day**, **month**, and **year** represent the current date under the Gregorian calendar as appropriate to the names of the fields, or a value whose field **DateValid** represents the value false and whose fields **day**, **month**, and **year** represent the date 'January 1, 1'. The field **month** shall have values such that the value for the month January is 1, the value for the month February is 2, and so forth, so that the value for the month December is 12.

Furthermore, the value attributed shall either have field **TimeValid** representing the value true, in which case fields **hour**, **minute**, and **second** shall represent the current time as appropriate to the names of the fields, or have field **TimeValid** representing the value false, in which case fields **hour**, **minute**, and **second** shall represent the time 'midnight' (0 hours, 0 minutes, 0 seconds).

The meaning of 'current date' and 'current time' shall be implementation-defined.

6.7.6 Required functions

6.7.6.1 General

The required functions shall be arithmetic functions, transfer functions, ordinal functions, Boolean functions, direct-access position functions, string functions, binding functions, and time functions.

6.7.6.2 Arithmetic functions

The types of operands and results for the required arithmetic functions shall be as shown in table 2. In all cases, x denotes the value of an expression, which is the operand referred to in table 2.

Table 2 — Arithmetic functions

Function	Result value	Type of operand	Type of result	Restriction
abs(x)	Absolute value (magnitude) of x	(1)	(5)	
sqr(x)	Square of x	(1)	(2)	(a)
sin(x)	Sine of x, x in radians	(1)	(3)	
cos(x)	Cosine of x, x in radians	(1)	(3)	
exp(x)	Base of natural logarithms raised to the power x	(1)	(3)	
ln(x)	Principal value of the natural logarithm of x	(1)	(3)	(b)
sqrt(x)	Principal value of the square root of x	(1)	(3)	(c)
arctan(x)	Principal value, in radians, of the arctangent of x	(1)	(3)	
arg(x)	Principal value, in radians, of the argument of x	(4)	Real-type	
re(x)	Real part of x	(4)	Real-type	
im(x)	Imaginary part of x	(4)	Real-type	
(1)	Integer-type, real-type, or complex-type			
(2)	The type of the result is the same type as that possessed by x			
(3)	If the type possessed by the operand is integer-type, the type of the result is real-type; otherwise, the type of the result is the same type as that possessed by x			
(4)	Complex-type			
(5)	If the type possessed by the operand is integer-type, the type of the result is integer-type; otherwise, the type of the result is real-type			
(a)	It shall be an error if no such value exists			
(b)	For x of integer-type or real-type, it shall be an error if x <= 0.0 For x of complex-type, it shall be an error if x = 0.0			
(c)	For x of integer-type or real-type, it shall be an error if x < 0.0			

NOTE — The principal value of the argument of x is greater than -*pi* and is less than or equal to *pi* (radians). The principal value of the natural logarithm of x has as its real part the natural logarithm of the absolute value of x, and as its imaginary part the principal value, in radians, of the argument of x. The principal value of the square root of x is the base of natural logarithms raised to the power one-half the principal value of the natural logarithm of x. Its argument is greater than -*pi*/2 and is less than or equal to *pi*/2 (radians); thus, its real part is non-negative. The principal value of the arctangent of x is (-*i*/2) times the principal value of the natural logarithm of (1+*i**x)/(1-*i**x), where *i* is the principal value of the square root of -1.

6.7.6.3 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}
```

card(x)

From the expression x that shall be of an unpacked-canonical-set-of-T-type or a packed-canonical-set-of-T-type, this function shall return a result of integer-type that shall equal the number of members of the value of the expression x. It shall be an error if no such value of integer-type exists.

cmplx(x,y)

From the expressions x and y that shall be of real-type, this function shall yield a result of complex-type. Cmplx(x,y) shall compute a complex value whose real part is an approximation to the value of x and whose imaginary part is an approximation to the value of y.

polar(r,t)

From the expressions r and t that shall be of real-type, this function shall yield a result of complex-type. Polar(r,t) shall compute a complex value whose magnitude is an approximation to the value of r and whose argument, in radians, is an approximation to the value of t.

6.7.6.4 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,k)

From the expression x that shall be of an ordinal-type and the expression k that shall be of integer-type, this function shall return a result that shall be of the ordinal-type. The function shall yield a value whose ordinal number is ord(x) + k, if such a value exists. It shall be an error if such a value does not exist.

succ(x)

Shall be equivalent to succ(x,1).

pred(x,k)

Shall be equivalent to succ(x,-(k)).

pred(x)

Shall be equivalent to succ(x,-1).

Examples:

```
{ The types shape and colour are defined in 6.4.10}
succ(yellow, -1)   { yields red}
succ(triangle, 0)  { yields triangle}
succ(yellow)       { yields green}
succ(yellow, 2)    { yields blue}
pred(red, -1)      { yields yellow}
pred(triangle, 0)  { yields triangle}
pred(green)        { yields yellow}
pred(blue, 2)      { yields yellow}
```

6.7.6.5 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**, which shall be implicitly accessible (see **6.11.4.2**) by the function-designator. 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 **6.4.3.6**); 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**, which shall be implicitly accessible (see **6.11.4.2**) by the function-designator. 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 **6.4.3.6**); otherwise, false.

empty(f)

The parameter f shall be a file-variable that possesses a direct-access file-type. When empty(f) is activated, it shall be an error if f is undefined; otherwise, the function shall yield the value true if f.L~f.R is the empty sequence (see **6.4.3.6**); otherwise, false.

6.7.6.6 Direct-access position functions

position(f)

The parameter f shall be a file-variable that possesses a direct-access file-type with index-type T. Let a be the smallest value of type T. When position(f) is activated, it shall be an error if f is undefined; otherwise, the function shall return a result of type T such that position(f) = succ(a, length(f.L)). It shall be an error if no such value exists.

LastPosition(f)

The parameter f shall be a file-variable that possesses a direct-access file-type with index-type T. Let a be the smallest value of type T. When LastPosition(f) is activated, it shall be an error if f is undefined; otherwise, the function shall return a result of type T such that LastPosition(f) = succ(a, length(f.L~f.R)-1). It shall be an error if no such value exists.

6.7.6.7 String functions

length(s)

From the expression s that shall be of char-type or a string-type, this function shall return a result of integer-type. The function shall yield the length of the value of s.

index(s1, s2)

From the expressions s1 and s2 that shall each be of char-type or a string-type, this function shall return a result of integer-type. If the value of s2 is the null-string, then the function shall yield 1; if the value of s1 is the null-string and the value of s2 is not the null-string, then the function shall yield 0; otherwise, letting s1v denote an auxiliary variable that the program does not otherwise contain and that possesses a variable-string-type with a capacity equal to the length of the value of s1 and letting the value attributed to s1v be the value of s1, the function shall yield the least i such that s1v[i..i+length(s2)-1] = s2, if such an i exists; otherwise, the function shall yield 0.

NOTE — 1 Index(s1,s2) determines whether string s1 contains string s2 as a substring. If s1 does not contain s2, then the value of index(s1,s2) is zero; otherwise, the value of index(s1,s2) is the index position of the first character position in s1 where a copy of s2 is located. The null-string is a substring of every string value located at index position 1.

substr(s, i, j)

From the expression s that shall be of char-type or a string-type and from the expressions i and j that shall be of integer-type, this function shall return a result of the canonical-string-type. It shall be an error if the value of i is less than or equal to 0. It shall be an error if the value of j is less than 0. It shall be an error if the value of (i)+(j)-1 is greater than the length of the value of s. Let sv denote an auxiliary variable that the program does not otherwise contain and that possesses a variable-string-type with a capacity equal to the greater of 1 and the length of the value of s. Let the value attributed to sv be the value of s. If the value of j equals 0, the function shall yield the null-string; otherwise, the function shall yield the value of sv[i..(i)+(j)-1].

NOTE — 2 Substr(s,i,j) computes the substring of string s beginning at position i and extending for length j.

substr(s, i)
Let sv denote an auxiliary variable that the program does not otherwise contain and that possesses a variable-string-type with a capacity equal to the greater of 1 and the length of the value of s. Let the value attributed to sv be the value of s. Let iv denote an auxiliary variable that the program does not otherwise contain and that possesses the integer-type. Let the value attributed to iv be the value of i. The function shall be equivalent to the expression substr(sv,iv,length(sv)-(iv)+1).

trim(s)
From the expression s that shall be of char-type or a string-type, this function shall return a result of the canonical-string-type. Let n be the length of the value of s. Let sv denote an auxiliary variable that the program does not otherwise contain and that possesses a variable-string-type with a capacity equal to the greater of 1 and n. Let the value attributed to sv be the value of s. If n equals 0, the function shall yield the null-string; if the value of sv[n] is not equal to the char-type value space, the function shall yield the value of sv; otherwise, the function shall yield the value of substr(sv,1,p-1), where p is the least value in the closed interval 1..n such that each component of sv[p..n] is the char-type value space.

For the following string comparison functions, the expressions s1 and s2 shall each be of char-type or the canonical-string-type. Let n1 be the length of the value of s1 and n2 be the length of the value of s2. Let s1v denote an auxiliary variable that the program does not otherwise contain and that possesses a variable-string-type with a capacity equal to the greater of 1 and n1. Let the value attributed to s1v be the value of s1. Let s2v denote an auxiliary variable that the program does not otherwise contain and that possesses a variable-string-type with a capacity equal to the greater of 1 and n2. Let the value attributed to s2v be the value of s2.

The result of each of the following string comparison functions shall be of Boolean-type.

EQ(s1,s2)
This function shall be equivalent to the expression

```
( (s1v = s2v) and (n1 = n2) ).
```

LT(s1,s2)
If n1 < n2, this function shall be equivalent to the expression

```
( s1v <= substr(s2v, 1, n1) );
```

otherwise, this function shall be equivalent to the expression

```
( substr(s1v, 1, n2) < s2v ) .
```

GT(s1,s2)
This function shall be equivalent to the expression

(not LT(s1v, s2v) and not EQ(s1v, s2v)) .

NE(s1,s2)

This function shall be equivalent to the expression

(not EQ(s1v, s2v)).

LE(s1,s2)

This function shall be equivalent to the expression

(LT(s1v, s2v) or EQ(s1v,s2v)) .

GE(s1,s2)

This function shall be equivalent to the expression

(not LT(s1v, s2v)).

NOTE — 3 It is possible for any of these functions to yield different results from their corresponding operators; for example, LT(a,b) could be false and a<b true.

6.7.6.8 Binding functions

binding(f)

The parameter f shall be a variable-access. The function shall access the variable denoted by f and shall return an implementation-defined value of the type denoted by the required type-identifier **BindingType** (see **6.4.3.4**). If the variable is bound to an external entity, the value of binding(f).bound shall be true; otherwise, the value of binding(f).bound shall be false. The value of binding(f) shall designate the status of the binding of the variable to an external entity. If the variable-access f possesses a file-type, it shall be a dynamic-violation if the variable does not possess the bindability that is bindable; otherwise, the variable shall possess the bindability that is bindable.

NOTES

1 Binding(f) is permitted even if f is totally-undefined.

2 Because the nature of external entities that might be bound to variables varies from processor to processor, the **BindingType** record-type may contain implementation-defined fields. The **binding** function allows a processor to provide initial values of type **BindingType** to a program without the program containing references to any of the implementation-defined fields. The **bound** field of the **BindingType** value returned by **binding** could be used by a program to test the success of an activation of the **bind** or **unbind** procedure. The **BindingType** value returned by **binding** can also be used to determine the result of any binding of program-parameters prior to activation of the main program (see **6.12**). The following example illustrates how the **binding** function may be used in this way.

Example:

```
procedure bindfile(var f : text);
   var
      b : BindingType;
   begin
      unbind(f);
      b := binding(f);
```

```
    repeat
      writeln('Enter file name:');
      readln(b.name);
      bind(f, b);
      b := binding(f);
      if not b.bound
      then
        writeln('File not bound--try again.');
    until b.bound;
end;
```

6.7.6.9 Time functions

date(t)
: From the expression t that shall be of the type denoted by the required type-identifier **TimeStamp**, this function shall return a result of the canonical-string-type with an implementation-defined length. The function shall yield a value that is an implementation-defined representation of the calendar date denoted by the value of t. It shall be an error if the fields **day**, **month**, and **year** of t do not represent a valid calendar date.

time(t)
: From the expression t that shall be of the type denoted by the required type-identifier **TimeStamp**, this function shall return a result of the canonical-string-type with an implementation-defined length. The function shall yield a value that is an implementation-defined representation of the time denoted by the value of t.

6.8 Expressions

6.8.1 General

An expression not contained by a schema-definition shall denote a value; an expression contained by a schema-definition shall denote a value for each tuple allowed by the actual-discriminant-part of the schema-definition. The use of a variable-access as a primary shall denote the value, if any, attributed to the variable accessed thereby. When a primary is used, it shall be an error if the variable denoted by a variable-access of the primary is undefined. Operator precedences shall be according to five classes of operators as follows. The operator not shall have the highest precedence, followed by the exponentiating-operators, followed by the multiplying-operators, 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 = primary [exponentiating-operator primary] .

primary > variable-access | unsigned-constant | set-constructor
| function-access | '(' expression ')' | 'not' primary
| constant-access | schema-discriminant
| structured-value-constructor | discriminant-identifier .

NOTE — 1 There is also a syntax rule for primary in **6.7.3.7.1**.

unsigned-constant = unsigned-number | character-string | 'nil' | extended-number .

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

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

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

```
x
15
(x + y + z)
sin(x + y)
[red, c, green]
[1, 5, 10..19, 23]
```

```
                          not p
                          pixel [red, c, green]
```

b) Factors:

```
                          x pow (-k)
                          x**y
```

c) Terms:

```
                          x * y
                          i / (1 - i)
                          (x <= y) and (y < z)
                          x*y**z
                          (x <> nil) and_then (x↑.field = 5)
```

d) Simple Expressions:

```
                          p or q
                          x + y
                          -x
                          hue1 + hue2
                          i * j + 1
                          x pow 3 + y pow 3 + z pow 3
                          (x = 0) or_else (a = (b/x))
```

e) Expressions:

```
                          x = 1.5
                          p <= q
                          p = q and r
                          (i < j) = (j < k)
                          c in hue1
                          x pow k > y pow k - z pow k
```

6.8.2 Constant-expressions

A constant-expression shall denote the value denoted by the expression of the constant-expression.

constant-expression = expression .

The expression of a constant-expression shall be nonvarying and shall not contain a discriminant-identifier.

An expression shall be designated *nonvarying* if it does not contain the following

a) an applied occurrence of an identifier as a variable-identifier, a schema-discriminant, a bound-identifier, or a field-designator-identifier; or

b) an applied occurrence of an identifier as a type-name that denotes a type that is not static; or

c) an applied occurrence of an identifier as a function-identifier that has a defining-point contained by the program-block or that denotes one of the required functions **eof** or **eoln**.

NOTES

1 By the above, it is implied that variable-accesses are excluded from constant-expressions. Similarly, the functions empty, position, and LastPosition cannot appear in constant-expressions because these functions require a variable as a parameter.

2 Since the accuracy of mathematical results of the real-type and of the complex-type are implementation-defined (see **6.4.2.2**), an implementation is required to specify the accuracy of constant-expressions.

3 See **6.3.2** for examples of the use of nonvarying expressions.

6.8.3 Operators

6.8.3.1 General

exponentiating-operator = '**' | 'pow' .

multiplying-operator = '*' | '/' | 'div' | 'mod' | 'and' | 'and_then' .

adding-operator = '+' | '−' | '><' | 'or' | 'or_else' .

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

A primary, a factor, a term, or a simple-expression shall be designated an *operand*. Except for the and_then and or_else operators, 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.

6.8.3.2 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 **6.8.3.4**), and the symbol + is also used as a string operator (see **6.8.3.6**).

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

A term of the form i div j shall be an error if j is zero; otherwise, the value of i div j shall be such that

abs(i) - abs(j) < abs((i div j) * j) <= abs(i)

where the value shall be zero if abs(i) < abs(j); otherwise, the sign of the value shall be positive if i and j have the same sign and negative if i and j have different signs.

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

Table 3 — Dyadic arithmetic operations

Operator	Operation	Type of operands	Type of result
+	Addition	(1)	(2)
-	Subtraction	(1)	(2)
*	Multiplication	(1)	(2)
/	Division	(1)	(3)
div	Division with truncation	Integer-type	Integer-type
mod	Modulo	Integer-type	Integer-type
**	Exponentiation to real power	(4)	(5)
pow	Exponentiation to integer power	(6)	Same as left operand

(1) Integer-type, real-type, or complex-type

(2) If at least one operand is of complex-type,
the type of the result is complex-type;
otherwise, if at least one operand is of real-type,
the type of the result is real-type;
otherwise, the type of the result is integer-type

(3) If at least one operand is of complex-type,
the type of the result is complex-type;
otherwise, the type of the result is real-type

(4) Left operand: integer-type, real-type, or complex-type;
right operand: integer-type or real-type;
in each case, if the operand is of integer-type, a real-type
approximation to its value is used

(5) If the left operand is of complex-type, the type of the result
is complex-type; otherwise, the type of the result is real-type

(6) Left operand: integer-type, real-type, or complex-type;
right operand: integer-type

NOTES

2 Only for i >= 0 and j > 0 does the relation (i div j) * j + i mod j = i hold.

3 See **6.4.2.2** for conditions under which the arithmetic operations are correctly performed.

A factor of the form x**y shall be an error if x is zero and y is less than or equal to zero.

A factor of the form x**y, where x is of integer-type or real-type, shall be an error if x is negative; otherwise, the value of x**y shall be zero if x is zero, else 1.0 if y is zero, else an approximation to (though not necessarily calculated by) exp(y*ln(x)).

The value of a factor of the form x**y, where x is of complex-type, shall be zero if x is zero, else 1.0 if y is zero, else an approximation to (though not necessarily calculated by) exp(y*ln(x)).

A factor of the form x pow y shall be an error if x is zero and y is less than or equal to zero.

The value of a factor of the form x pow y, where x is of integer-type, shall be zero if x is zero, else 1 if y is zero, else equal to x*(x pow (y-1)) if y is positive, else equal to (1 div x) pow (-y) if y is

Table 4 — Monadic arithmetic operations

Operator	Operation	Type of operand	Type of result
+	Identity	(1)	Same as operand
-	Sign-inversion	(1)	Same as operand
(1)	Integer-type, real-type, or complex-type		

negative.

The value of a factor of the form x pow y, where x is of real-type or complex-type, shall be zero if x is zero, else 1.0 if y is zero, else an approximation to x*(x pow (y-1)) if y is positive, else an approximation to (1/x) pow (-y) if y is negative.

6.8.3.3 Boolean operators

Operands and results for Boolean operations shall be of Boolean-type. The Boolean operators or, or_else, and, and_then, and not shall denote respectively the logical operations of disjunction, disjunction, conjunction, conjunction, and negation. In a term of the form A and_then B, the right operand shall be evaluated if and only if the left operand denotes the value true; the term shall denote the value false if the left operand denotes the value false; otherwise, the term shall denote the value denoted by the right operand. In a simple-expression of the form A or_else B, the right operand shall be evaluated if and only if the left operand denotes the value false; the simple-expression shall denote the value true if the left operand denotes the value true; otherwise, the simple-expression denotes the value denoted by the right operand.

In a term of the form A and_then B, the right operand shall not be in error if the left operand denotes the value false. In a simple-expression of the form A or_else B, the right operand shall not be in error if the left operand denotes the value true.

Boolean-expression = expression .

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

6.8.3.4 Set operators

The types of operands and results for set operations shall be as shown in table 5.

Table 5 — Set operations

Operator	Operation	Type of operands	Type of result
+	Set union	(1)	Same as the operands
-	Set difference	(1)	Same as the operands
*	Set intersection	(1)	Same as the operands
><	Set symmetric difference	(1)	Same as the operands
(1)	The same unpacked-canonical-set-of-T-type or packed-canonical-set-of-T-type (see **6.8.1**)		

Where x denotes a value of the ordinal-type T and u and v are operands of an unpacked-canonical-set-of-T-type or a packed-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;

— 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 or is a member of the value of v and not a member of the value of u.

6.8.3.5 Relational operators

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

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 string-type or any simple-type except complex-type	Boolean-type
`<= >=`	Any string-type, unpacked-canonical-set-of-T-type, packed-canonical-set-of-T-type, or any simple-type except complex-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

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, or one operand shall be of complex-type and the other shall be either of real-type or 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 — 1 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 **6.4.5**), they shall denote the lexicographic relations defined below. This lexicographic ordering shall impose a total ordering on values of a string-type.

Let s1 and s2 be two values of compatible string-types where the length of s1 is less than or equal to the length of s2, let n1 be the length of s1, and let n2 be the length of s2; then

```
s1 = s2  iff (for all i in [1..n1]: s1[i] = s2[i])
             and (for all i in [n1+1..n2]: ' ' = s2[i])

s1 < s2  iff ( there exists p in [1..n1]:
               (for all i in [1..p-1]: s1[i] = s2[i])
               and s1[p] < s2[p] )
                         or
             ( (for all i in [1..n1]: s1[i] = s2[i] )
               and ( there exists p in [n1+1..n2]:
                     (for all i in [n1+1..p-1]: ' ' = s2[i])
                     and ' ' < s2[p]) )
```

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

The definitions of the relational operators for the length of s1 greater than the length of s2 are derived from the definitions of the operators for the length of s1 less than or equal to the length of s2.

When comparing a char-type value with a string-type value, the char-type value shall be treated as a value of the canonical-string-type with length 1 and with the component-value equal to the char-type value.

NOTES

2 For comparison of values of compatible char-types or string-types, the relational-operators effectively extend the shorter value with trailing spaces to the length of the longer value.

3 String-type ordering is defined in terms of the char-type ordering, in turn defined in table 6.

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.

6.8.3.6 String operator

The types of operands and results for the string operator shall be as shown in table 7.

Table 7 — String operation

Operator	Operation	Type of operands	Type of result
+	String concatenation	Char-type or the canonical-string-type	Canonical-string-type

Where a and b denote operands possessing the char-type or the canonical-string-type, $a + b$ shall denote a value of the canonical-string-type whose length shall be equal to the sum of the length of a and the length of b. The value of the components of $a + b$ in order of increasing index shall be the values of the components of a in order of increasing index or the char-type value of a, followed by the values of the components of b in order of increasing index or the char-type value of b.

6.8.4 Schema-discriminants

schema-discriminant = (variable-access | constant-access) '.' discriminant-specifier
| schema-discriminant-identifier .

discriminant-specifier = discriminant-identifier .

If a schema-discriminant closest-contains a variable-access or constant-access, the variable-access or constant-access shall possess a type produced from a schema with a tuple, and the schema-discriminant shall possess the type possessed by, and denote the value corresponding to, the discriminant-identifier of the discriminant-specifier of the schema-discriminant according to the tuple. If a schema-discriminant closest-contains a schema-discriminant-identifier, the schema-discriminant shall possess the type possessed by, and denote the value denoted by, the schema-discriminant-identifier. The occurrence of the variable-access or constant-access shall constitute the defining-point of each of the discriminant-identifiers that is a formal discriminant of the schema for the region that is the discriminant-specifier of the schema-discriminant.

Examples:

```
ShowScreen.height
ShowScreen.width
MyVector.vlength
```

6.8.5 Function-designators

A function-designator shall specify the activation of the block of the function-block of the function (see **6.2.3.2 j)**) denoted by the function-name 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 **6.9.2.4**), 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, the function-designator shall contain 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 **6.7.3**. The order of evaluation, accessing, and binding of the actual-parameters shall be implementation-dependent.

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

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

actual-parameter = expression | variable-access
| procedure-name | function-name .

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

6.8.6 Function-accesses

6.8.6.1 General

A function-access, according to whether it is an entire-function-access, a component-function-access, or a substring-function-access, shall denote the value of the result of an activation, a component of the value of another function-access, or a substring of the value of another function-access, respectively. The value and type of a function-access shall be the value and type, respectively, either of the entire-function-access or substring-function-access of the function-access, or of the indexed-function-access or record-function-access of the component-function-access of the function-access.

function-access = entire-function-access
| component-function-access
| substring-function-access .

component-function-access = indexed-function-access
| record-function-access .

entire-function-access = function-designator .

An entire-function-access shall denote the value of the result of the activation of the block of the function denoted by the function-name of the function-designator of the entire-function-access.

NOTE — A function-access is not equivalent to a variable-access. For example, a function-access may not be used as an actual variable parameter or as the record-variable in a with-statement.

6.8.6.2 Indexed-function-accesses

An indexed-function-access shall denote a component of the value of a function-access possessing an array-type or a string-type.

indexed-function-access = array-function '[' index-expression { ',' index-expression } ']'
| string-function '[' index-expression ']' .

array-function = function-access .

string-function = function-access .

An array-function shall be a function-access possessing an array-type. A string-function shall be a function-access possessing a string-type. The string-function of an indexed-function-access shall be a function-access possessing a variable-string-type.

NOTE — Function-accesses possessing a fixed-string-type are indexed using array-type properties.

For an array-function in an indexed-function-access closest-containing a single index-expression, the value of the index-expression shall be assignment-compatible with the index-type of the array-type possessed by the array-function.

For a string-function in an indexed-function-access, the index-expression of the indexed-function-access shall possess the integer-type, and it shall be an error if the value of the index-expression is not in the index-domain of the value of the string-function.

The component denoted by the indexed-function-access shall be the component that corresponds to the value of the index-expression by the mapping of the type possessed by the array-function (see **6.4.3.2**) or string-function (see **6.4.3.3**).

If the array-function or string-function is itself an indexed-function-access, 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 evaluation both of the index-expressions of, and of the array-function or string-function of, an indexed-function-access shall be implementation-dependent.

6.8.6.3 Record-function-accesses

A record-function-access shall denote that component of the value of the record-function of the record-function-access associated (see **6.4.3.4**) with the field-identifier of the field-specifier of the record-function-access. A record-function shall be a function-access possessing a record-type.

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

record-function-access = record-function '.' field-specifier .

record-function = function-access .

It shall be an error to denote a component of a variant, unless the variant is active.

6.8.6.4 Function-identified-variables

A function-identified-variable shall denote the variable identified by the value of the pointer-function of the function-identified-variable. A pointer-function shall be a function-access possessing a pointer-type.

function-identified-variable = pointer-function '↑' .

pointer-function = function-access .

It shall be an error if the pointer-function of a function-identified-variable denotes the nil-value.

6.8.6.5 Substring-function-accesses

A substring-function-access shall denote a value of the canonical-string-type.

substring-function-access = string-function '[' index-expression '..' index-expression ']' .

The index-expressions in a substring-function-access shall possess the integer-type. It shall be an error if the value of an index-expression in a substring-function-access is less than one or greater than the length of the value of the string-function of the substring-function-access or if the value of the first index-expression is greater than the value of the second index-expression. The length of the string-type value of the substring-function-access shall be equal to one plus the value of the second index-expression minus the value of the first index-expression. The components of the value of the substring-function-access shall be, in order of increasing index, the contiguous components of the value of the string-function from the component that corresponds to the value of the first index-expression through the component that corresponds to the value of the second index-expression.

The order of evaluation both of the index-expressions of, and of the string-function of, a substring-function-access shall be implementation-dependent.

6.8.7 Structured-value-constructors

6.8.7.1 General

A structured-value-constructor shall denote a value of the type of the structured-value-constructor. That type shall be a type that is permissible as the component-type of a file-type (see **6.4.3.6**). The order of evaluation of the component-values contained by a structured-value-constructor shall be implementation-dependent.

structured-value-constructor = array-type-name array-value
| record-type-name record-value
| set-type-name set-value .

component-value = expression | array-value | record-value .

The type of a structured-value-constructor shall be the type denoted by the array-type-name, record-type-name, or set-type-name of the structured-value-constructor. The type of an array-value, a record-value, or a set-value of either a structured-value-constructor or a component-value shall be the type of the structured-value-constructor or the component-value, respectively. The value denoted by an expression in a component-value shall be assignment-compatible with the type of the component-value. The structure of a value possessing a structured-type shall be the structure of the structured-type.

6.8.7.2 Array-values

The type of an array-value shall be an array-type, and the array-value shall denote a value of that type.

array-value = '[' [array-value-element { ';' array-value-element } [';']]
[array-value-completer [';']] ']' .

array-value-element = case-constant-list ':' component-value .

array-value-completer = 'otherwise' component-value .

The type of a component-value of either an array-value-element or an array-value-completer of an array-value shall be the component-type of the array-type of the array-value.

The values denoted by the case-ranges of the case-constant-lists of the array-value-elements of an array-value shall be distinct and shall belong to the set of values determined by the index-type of the array-type possessed by the array-value. Every component of an array-value shall be a value, as specified by one of the following two statements.

a) The component mapped to by each value denoted by a case-range of a case-constant-list of an array-value-element of the array-value shall be the value denoted by the component-value of the array-value-element.

b) Any component not mapped to by a value denoted by a case-range of a case-constant-list of an array-value-element of the array-value shall be the value denoted by the component-value of the array-value-completer of the array-value. If there is at least one such component, there shall be an array-value-completer in the array-value.

NOTE — Consequently, every component of the array-value must be specified.

6.8.7.3 Record-values

The type of a record-value shall be a record-type, and the record-value shall denote a value of that type.

record-value = '[' field-list-value ']' .

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

fixed-part-value = field-value { ';' field-value } .

field-value = field-identifier { ',' field-identifier } ':' component-value .

variant-part-value = 'case' [tag-field-identifier ':']
constant-tag-value 'of' '[' field-list-value ']' .

constant-tag-value = constant-expression .

tag-field-identifier = field-identifier .

The occurrence of a record-value shall constitute the defining-point of each of the field-identifiers of the record-type of the record-value as field-identifiers associated with the components of the record-value for each region that is a field-identifier closest-contained by the record-value. The component associated with each field-identifier in a field-value shall be the value denoted by the component-value of that field-value. The type of the component-value of a field-value shall be the type of each of the components that are components of the record-type of the record-value closest-containing the field-value and that are associated with the field-identifiers of the field-value.

NOTE — 1 Consequently, all field-identifiers in a field-value must have been declared to have the same type.

Each field-identifier in a field-value of a fixed-part-value of a field-list-value that corresponds to a field-list shall denote a field of the field-list. The field-list-value of a record-value shall correspond to the field-list of the record-type possessed by the record-value. The fixed-part-value or variant-part-value of a field-list-value shall correspond to the fixed-part or variant-part, respectively, of the field-list corresponding to the field-list-value. The constant-expression of a constant-tag-value of a variant-part-value shall denote a value belonging to the set of values determined by the variant-type of the variant-part corresponding to the variant-part-value. The field-list-value of a variant-part-value shall correspond to the field-list of the variant corresponding to the value of the constant-expression of the constant-tag-value of the variant-part-value; the selector component of the variant-part-value shall be a value that is associated with that variant. A tag-field-identifier in a variant-part-value shall be the field-identifier associated with the selector of the variant-part corresponding to the variant-part-value; the component of the variant-part-value associated with the field-identifier shall be the selector of the variant-part and shall be the value denoted by the constant-tag-value of the variant-part-value. The field-identifier, if any, associated with the selector of a variant-part shall have an applied occurrence in the tag-field-identifier of each variant-part-value corresponding to the variant-part.

For each field-list-value that corresponds to a field-list, each field-identifier associated with a component of the field-list shall have exactly one applied occurrence as a field-identifier closest-contained by the field-list-value.

NOTE — 2 Consequently, every component of the record-value, including each active variant, must be specified as a value. Also, a field-identifier cannot be specified more than once in a record-value.

6.8.7.4 Set-values

The type of a set-value shall be a set-type, and the set-value shall denote a value of that type.

set-value = set-constructor .

The value of the set-constructor of a set-value shall be assignment-compatible with the type of the set-value.

6.8.8 Constant-accesses

6.8.8.1 General

NOTE — Neither a constant-access nor a constant-access-component is necessarily a constant. For example, given the following declarations

```
   t = array [1..3] of integer;
const
   c = t[1:1; 2:2; 3:3];
var
   i: integer;
```

and the following code segment

```
for i := 1 to 3 do
  writeln(c[i]);
```

the constant-access, c[i], denotes a different value for each iteration of the loop.

A constant-access-component shall denote a component or a substring of a value.

constant-access = constant-access-component | constant-name .

constant-access-component = indexed-constant
| field-designated-constant
| substring-constant .

The value and type of a constant-access shall be the value and type, respectively, either of the constant-name of the constant-access or of the indexed-constant, field-designated-constant, or substring-constant of the constant-access-component.

6.8.8.2 Indexed-constants

An indexed-constant shall denote a component of a value possessing an array-type or a string-type.

indexed-constant = array-constant '[' index-expression { ',' index-expression } ']'
| string-constant '[' index-expression ']' .

array-constant = constant-access .

string-constant = constant-access .

An array-constant shall be a constant-access possessing an array-type. A string-constant shall be a constant-access possessing a string-type. The string-constant of an indexed-constant shall be a constant-access possessing a variable-string-type.

NOTE — Constant-accesses possessing a fixed-string-type are indexed using array-type properties.

For an array-constant in an indexed-constant closest-containing a single index-expression (see **6.5.3.2**), the value of the index-expression of the indexed-constant shall be assignment-compatible with the index-type of the array-type of the array-constant.

For a string-constant in an indexed-constant, the index-expression of the indexed-constant shall possess the integer-type, and it shall be an error if the value of the index-expression is not in the index-domain of the value of the string-constant.

The component denoted by the indexed-constant shall be the component that corresponds to the value of the index-expression by the mapping of the type possessed by the array-constant (see **6.4.3.2**) or string-constant (see **6.4.3.3**).

If the array-constant is itself an indexed-constant, 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 evaluation of the index-expressions of an indexed-constant shall be implementation-dependent.

Examples:

```
UnitVector[limit]
BlankCard[1]
```

6.8.8.3 Field-designated-constants

A field-designated-constant either shall denote that component of the value denoted by the record-constant of the field-designated-constant associated (see **6.4.3.4**) with the field-identifier of the field-specifier (see **6.5.3.3**) of the field-designated-constant or shall denote the value denoted by the constant-field-identifier (see **6.9.3.10**) of the field-designated-constant.

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

field-designated-constant = record-constant '.' field-specifier
| constant-field-identifier .

record-constant = constant-access .

A record-constant shall be a constant-access possessing a record-type.

It shall be an error to denote a component of a variant, unless the variant is active.

Examples:

```
origin.r
origin.theta
unit.theta
```

6.8.8.4 Substring-constants

A substring-constant shall denote a value of the canonical-string-type.

substring-constant = string-constant '[' index-expression '..' index-expression ']' .

The index-expressions in a substring-constant shall possess the integer-type. It shall be an error if the value of an index-expression in a substring-constant is less than 1 or greater than the length of the value of the string-constant of the substring-constant or if the value of the first index-expression is greater than the value of the second index-expression. The length of the string-type value of the substring-constant shall be equal to one plus the value of the second index-expression minus the value of the first index-expression. The components of the value of the substring-constant shall be, in order of increasing index, the contiguous components of the value of the string-constant from the component that corresponds to the value of the first index-expression through the component that corresponds to the value of the second index-expression.

The order of evaluation of the index-expressions of a substring-constant shall be implementation-dependent.

Example:
```
hex_string[14..16]
```

6.9 Statements

6.9.1 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 **6.9.2.4**) 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.

6.9.2 Simple-statements

6.9.2.1 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 = .

6.9.2.2 Assignment-statements

An assignment-statement shall attribute the value of the expression of the assignment-statement to the variable that is denoted by the variable-access of the assignment-statement or that is the result of the activation of the function denoted by the function-identifier of the assignment-statement. The value shall be assignment-compatible with the type of the variable denoted by the variable-access, or the underlying-type (see **6.4.2.5**) of the type of the variable that is the result of the activation. The function-block associated (see **6.7.2**) 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.

Examples:

```
x := y + z
p := (1 <= i) and (i < 100)
i := sqr(k) - (i * j)
hue1 := [blue, succ(c)]
p1↑.mother := true
full_name := last_name + ', ' + first_name + ' ' + middle_initial
               + '., ' + mister {'Grant, Ulysses S., Mr.'}
```

6.9.2.3 Procedure-statements

A procedure-statement shall specify the activation of the block of the procedure-block of the procedure (see **6.2.3.2 i)**) denoted by the procedure-name 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 **6.7.3**.

The order of evaluation, accessing, and binding of the actual-parameters shall be implementation-dependent.

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

procedure-statement = procedure-name ([actual-parameter-list]
| read-parameter-list | readln-parameter-list | readstr-parameter-list
| write-parameter-list | writeln-parameter-list | writestr-parameter-list) .

Examples:

```
PrepareForAppending(f)
halt
```

6.9.2.4 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;

b) any activation containing the activation-point of an activation required by exceptions a) or b) not to be terminated; and

c) each of the activations that comprise the activation of the program-block (see **6.2.3.6**).

goto-statement = 'goto' label .

It shall be a dynamic-violation if the commencement of the activation containing the program-point has not completed (see **6.2.3.8**).

6.9.3 Structured-statements

6.9.3.1 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.

6.9.3.2 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
```

6.9.3.3 Conditional-statements

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

6.9.3.4 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 )
```

6.9.3.5 Case-statements

The case-index of a case-statement and each case-constant closest-contained by the case-constant-list of a case-list-element of the case-statement shall all possess the same ordinal-type; no value shall be denoted by more than one case-range closest-contained by the case-constant-list of any case-list-elements of the case-statement. On execution of the case-statement, the case-index shall be evaluated. If a case-range closest-contained by a case-constant-list of a case-list-element of the case-statement denotes that value, the statement of the case-list-element shall be executed; otherwise, if a case-statement-completer occurs in the case-statement, the statement-sequence of the case-statement-completer shall be executed; otherwise, it shall be a dynamic-violation.

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

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

case-index = expression .

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

case-statement-completer = 'otherwise' statement-sequence .

Examples:

```
1) case operator of
```

```
    plus:   i := i + j;
    minus:  i := i - j;
    times:  i := i * j;
    divvy:  case j of
             -maxint..-1, 1..maxint: i := i div j;
             0 : begin
                  writeln('divide by zero!');
                  halt;
                 end
             otherwise i := 0; writeln(' See 6.4.2.2 a).')
            end
   end

2) if limit >= 0
   then
     case i of
      -maxint..(-limit-1):   writeln('too small');
      -limit..limit:         writeln('just right');
      (limit+1)..maxint:     writeln('too big')
     end
   else
    writeln('limit is less than 0');
```

6.9.3.6 Repetitive-statements

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

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

6.9.3.7 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
```

6.9.3.8 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
```

6.9.3.9 For-statements

6.9.3.9.1 General

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 iteration-clause 'do' statement .

control-variable = entire-variable .

iteration-clause = sequence-iteration | set-member-iteration .

The control-variable shall be an entire-variable whose identifier is declared in a variable-declaration-part of the block closest-containing the for-statement. The control-variable shall possess an ordinal-type and shall be nonbindable. 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 (see **6.9.4**) a variable-access denoting the variable denoted by the control-variable of the for-statement.

6.9.3.9.2 Sequence-iteration

sequence-iteration = ':=' initial-value ('to' | 'downto') final-value .

initial-value = expression .

final-value = expression .

The initial-value and the final-value of a sequence-iteration of an iteration-clause of a for-statement shall be of a type compatible with the type of the control-variable of the for-statement. 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.

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 range-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)
```

6.9.3.9.3 Set-member-iteration

set-member-iteration = 'in' set-expression .

set-expression = expression .

The set-expression of a set-member-iteration of an iteration-clause of a for-statement shall possess an unpacked-canonical-set-of-T-type or a packed-canonical-set-of-T-type. The type of the control-variable of the for-statement shall be compatible with T. The set-expression shall be evaluated prior to the first execution, if any, of the statement of the for-statement. Each value, if any, that is a member of the value of the set-expression shall be assignment-compatible with the type possessed by the control-variable. For each member of the value of the set-expression, the value that is the member shall be attributed to the control-variable, and then the statement of the for-statement shall be executed. The order in which members of the value of the set-expression are selected shall be implementation-dependent.

Examples:

```
1)   for c in hue1 do q(c)
```

```
2)   for status in DeviceStatus do
        case status of
          Busy:
             { respond to Busy };
          ParityError:
             { respond to ParityError };
          OutOfPaper:
             { respond to OutOfPaper };
          LineBreak:
             { respond to LineBreak }
        end
```

6.9.3.10 With-statements

with-statement = 'with' with-list 'do' statement .

with-list = with-element { ',' with-element } .

with-element = variable-access | constant-access .

field-designator-identifier = identifier .

constant-field-identifier = identifier .

schema-discriminant-identifier = identifier .

A with-statement shall specify the execution of the statement of the with-statement. The constant-access or variable-access of a with-element shall possess either a type produced from a schema or a record-type. The occurrence of a variable-access or constant-access, that possesses a record-type, in the only with-element in the with-list of a with-statement shall constitute the defining-point of each of the field-identifiers associated with components of the record-type as a field-designator-identifier or constant-field-identifier, respectively, for the region that is the statement of the with-statement; each applied occurrence of the field-designator-identifier or constant-field-identifier shall denote that component, either of the variable denoted by the variable-access or of the value denoted by the constant-access, respectively, that is associated with the field-identifier by the record-type.

An occurrence of a variable-access or constant-access, that possesses a type produced from a schema with a tuple, in the only with-element in the with-list of a with-statement shall constitute the defining-point of each discriminant-identifier that is a formal discriminant of the schema as a schema-discriminant-identifier for the region that is the statement of the with-statement; each applied occurrence of the schema-discriminant-identifier shall possess the type possessed by the discriminant-identifier and shall denote the value corresponding to the discriminant-identifier according to the tuple.

The variable-access shall be accessed or the value of the constant-access shall be determined before the statement of the with-statement is executed, and the access to the variable 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
```

Examples:

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

{ has the same effect on the variable Good_thru as

      if Good_thru.month = 12
      then begin
           Good_thru.month := 1;
           Good_thru.year := Good_thru.year+1
           end
      else Good_thru.month := Good_thru.month+1;          }

2) with ShowScreen do
      if (width = 80) and (height = 24)
        then  { write full screen }
        else  { write line by line }

3) with unit do
      begin
        x := r;
        coordinate.theta := theta
      end;
```

NOTE — Month and year in Example 1) are field-designator-identifiers, width and height in Example 2) are schema-discriminant-identifiers, and r and theta in Example 3) are constant-field-identifiers.

6.9.4 Threats

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

a) S is an assignment-statement and V is in S.

b) S contains V in an actual-parameter that is an actual variable parameter corresponding to a formal variable parameter that is not protected (see **6.7.3.1**).

c) S is a procedure-statement that specifies the activation of one of the required procedures **read**, **readln**, or **readstr**, and V is either in an actual-parameter of an actual-parameter-list of S or in a read-parameter-list, a readln-parameter-list, or a readstr-parameter-list of S, respectively.

d) S is a procedure-statement that specifies activation of the required procedure **writestr**, and V is in the string-variable accessed by the activation.

e) S is a procedure-statement that specifies activation of the required procedure **new**, and V is the variable-access p (see **6.7.5.3**).

f) S is a procedure-statement that specifies activation of the required procedure **GetTimeStamp**, and V is the variable-access t (see **6.7.5.8**).

g) S is a for-statement and V denotes the control-variable of S.

h) V is in an array-variable, record-variable, or string-variable, and S is threatening a variable-access closest-containing V.

i) S is a with-statement, V is in a with-element in the with-list of S, and S contains a statement threatening a variable-access closest-containing a field-designator-identifier having V as a defining-point.

j) S is a procedure-statement that specifies the activation of the required procedure **bind** or **unbind**, and V is the variable-access f (see **6.7.5.6**).

NOTE — In **6.7.5.4**, the execution of the required procedures **pack** and **unpack** is defined as equivalent to a series of assignments of the components of the packed and unpacked arrays. These equivalent assignments are subject to a) and i) above.

6.10 Input and output

6.10.1 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**, which shall be implicitly accessible (see **6.11.4.2**) by the procedure-statement.

The following 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 **6.7.5.2**. 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)).

NOTES

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

2 The sequence r represents the initial spaces and end-of-lines skipped during reading; s represents the quantity read; t~u represents text remaining to be read; and t represents the largest prefix of t~u that does not contain an end-of-line.

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; v_1,...,v_n shall be variable-accesses, each of which shall possess 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 — 3 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~u).first) shall not, form a signed-integer according to the syntax of **6.1.7**. 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 — 4 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~u).first) shall not, form a number according to the syntax of **6.1.7**. 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 — 5 The sequence r represents any spaces and end-of-lines to be skipped, and the sequence s represents the number to be read.

e) If v is a variable-access possessing a fixed-string-type of capacity c, read(f,v) shall satisfy the following requirements. Length(r) shall equal 0, no component of s shall equal end-of-line, and the remaining execution of the statement shall cause a value to be attributed to v. That value shall be the value of the fixed-string-type whose components in order of increasing index consist of the components of s, in order, followed by zero or more spaces. If c exceeds length(s~t), length(t) shall equal 0; otherwise, length(s) shall equal c.

NOTE — 6 If eoln(f) is initially true, then no characters are read, and the value of each component of v is a space.

f) If v is a variable-access possessing a variable-string-type of capacity c, read(f,v) shall satisfy the following requirements. Length(r) shall equal 0, no component of s shall equal end-of-line, and the remaining execution of the statement shall cause a value to be attributed to v. That value shall be the value of the variable-string-type whose components in order of increasing index consist of the components of s, in order. If c exceeds length(s~t), length(t) shall equal 0; otherwise, length(s) shall equal c.

NOTE — 7 If eoln(f) is initially true, then no characters are read, and the value of v is the null-string.

6.10.2 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**, which shall be implicitly accessible (see **6.11.4.2**) by the procedure-statement.

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 **6.7.5.2**).

6.10.3 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**, which shall be implicitly accessible (see **6.11.4.2**) by the procedure-statement. When **write** is applied to a textfile f, it shall be an error if f is undefined or f.M = Inspection (see **6.4.3.6**).

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 **6.7.5.2**).

6.10.3.1 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 designated the *field-width parameters*. The value of TotalWidth shall be greater than or equal to zero; it shall be an error if the value is less than zero. The value of FracDigits shall be greater than or equal to zero; it shall be an error if the value is less than zero.

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 **6.10.3.4.2**).

6.10.3.2 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

```
if TotalWidth > 0,
  (TotalWidth − 1) spaces, the character value of e;

if TotalWidth=0,
  no characters.
```

6.10.3.3 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 pow (z−1) ≤ abs(x) < 10 pow 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).

6.10.3.4 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.

6.10.3.4.1 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 RealSize ( y : real ) : integer ;
 { Returns the value, z, such that 10.0 pow (z−1) <= abs(y) < 10.0 pow 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 / 10.0 pow ExpValue;
     eWritten := eWritten + 0.5 * 10.0 pow(−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).
```

6.10.3.4.2 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 RealSize and Truncate described in **6.10.3.4.1**;

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 * 10.0 pow(−FracDigits);
    eWritten := Truncate ( eWritten, FracDigits )
  end;
```

let IntDigits be the positive integer defined by

```
if 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.

6.10.3.5 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 **6.10.3.6**), where the case of each letter is implementation-defined, with a field-width parameter of TotalWidth.

6.10.3.6 String-types

If the value of e is a string-type value with a length of n, the default value of TotalWidth shall be n. The representation shall consist of

if TotalWidth > n,
  (TotalWidth − n) spaces,
if n > 0,
  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.
if TotalWidth = 0,
  no characters.

6.10.4 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**, which shall be implicitly accessible (see **6.11.4.2**) by the procedure-statement.

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,p_1,...,p_n); 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 **6.7.5.2**.

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(end-of-line) is the sequence consisting solely of the end-of-line component defined in **6.4.3.6**.

NOTE — Writeln(f) terminates the partial line, if any, that is being generated. By the conventions of **6.7.5.2** it is an error if the pre-assertion is not true prior to writeln(f).

6.10.5 The procedure page

It shall be an error if the pre-assertion required for writeln(f) (see **6.10.4**) 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**, which shall be implicitly accessible (see **6.11.4.2**) by the procedure-statement. 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 **6.4.3.6**), 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.

6.11 Modules

6.11.1 Module-declarations

module-declaration = module-heading [';' module-block]
| module-identification ';' module-block .

module-heading = 'module' identifier [interface-directive]
['(' module-parameter-list ')'] ';'
interface-specification-part
import-part
{ constant-definition-part
| type-definition-part
| variable-declaration-part
| procedure-and-function-heading-part }
'end' .

module-parameter-list = identifier-list .

procedure-and-function-heading-part = (procedure-heading | function-heading) ';' .

module-identification = 'module' module-identifier implementation-directive .

module-identifier = identifier .

module-block = import-part
{ constant-definition-part
| type-definition-part
| variable-declaration-part
| procedure-and-function-declaration-part }
[initialization-part]
[finalization-part]
'end' .

initialization-part = 'to' 'begin' 'do' statement ';' .

finalization-part = 'to' 'end' 'do' statement ';' .

The occurrence of an identifier in the module-heading of a module-declaration shall constitute its defining-point as a module-identifier for each region that is either the identifier of a module-heading contained by the program or the module-identifier of a module-identification contained by the program.

NOTE — 1 The module-identifier has meaning only in places where a module-identifier is either defined or referenced. A module-identifier does not otherwise affect the program.

The occurrence of a module-block in a module-declaration that contains a module-heading shall *associate* that module-block with that module-heading. The occurrence of a module-block in a module-declaration that contains a module-identification shall *associate* that module-block with the module-heading containing the defining-point of the module-identifier of that module-identification. There shall be exactly one module-block associated with a module-heading. A module-block together with its associated module-heading shall constitute a module, and each shall be said to be *associated* with that module.

An interface-directive shall occur in a module-heading of a module-declaration if and only if a module-block does not occur in the module-declaration.

Each identifier having a defining-point as a module-identifier in a module-heading of a module-declaration containing the interface-directive interface shall have exactly one of its applied occurrences in a module-identification of a module-declaration containing the implementation-directive implementation. These two module-declarations shall both be program-components of the program-block (see **6.13**).

For any two distinct modules A and B such that A supplies B and B supplies A, neither the module-block of A nor the module-block of B shall contain an initialization-part; neither module-block shall contain a finalization-part; and an expression contained by the module-heading of either A or B shall be nonvarying (see **6.8.2**).

NOTES

2 This can happen, for example, when the module-heading of A exports an interface that is imported by the module-block, but not the module-heading, of B; and the module-heading of B exports an interface that is imported by the module-heading or module-block of A.

3 Modules may directly or indirectly supply each other. For example, if A supplies B and B supplies C and C supplies A, then none of the three modules can have an initialization-part or a finalization-part, and any discriminant-values and subrange-bounds in their module-headings must be nonvarying.

The identifiers contained by the module-parameter-list of a module-heading shall have distinct spellings, and for each such identifier there shall be a defining-point as a variable-identifier with the same spelling for the region that is the module-heading. If the spelling is neither **input** nor **output**, the variable-identifier either shall be local to the module or shall be an imported variable-identifier that is a module-parameter. If the spelling is **input** or **output**, the occurrence of the identifier contained by the module-parameter-list shall constitute a defining-point for the region that is the module-heading as a variable-identifier denoting the required textfile **input** or **output**, respectively. If the variable-identifier is local to the module or has the spelling **input** or **output**, both the variable-identifier and any variable it denotes shall be designated a *module-parameter*. The binding of a variable that is a module-parameter to entities external to the program shall be implementation-defined.

NOTES

4 The external representation of external entities bound to module-parameters is not defined by this International Standard.

5 Furthermore, two different modules may specify that two different variables whose variable-identifiers have the same spelling are to be bound to external entities — this International Standard does not specify whether such variables are to be bound to the same external entity or to different external entities.

6 Variables that are module-parameters are not necessarily bound when the module is activated.

6.11.2 Export-part

An export-part shall introduce an identifier to denote an interface. An export-list shall introduce one or more constituent-identifiers.

interface-specification-part = 'export' export-part ';' { export-part ';' } .

export-part = identifier '=' '(' export-list ')' .

export-list = (export-clause | export-range) { ',' (export-clause | export-range) } .

export-clause = exportable-name | export-renaming-clause .

export-renaming-clause = exportable-name '=>' identifier .

exportable-name = constant-name
| type-name
| schema-name
| ['protected'] variable-name
| procedure-name
| function-name .

export-range = first-constant-name '..' last-constant-name .

first-constant-name = constant-name .

last-constant-name = constant-name .

constituent-identifier = identifier .

interface-identifier = identifier .

The occurrence of an identifier in an export-part shall constitute its defining-point as an interface-identifier for each region that is either the identifier of an export-part contained by the program-block or the interface-identifier of an import-specification contained by the program-block.

The occurrence of an exportable-name in an export-clause shall constitute the defining-point of the identifier of the constant-identifier, type-identifier, schema-identifier, variable-identifier, procedure-identifier, or function-identifier contained by the exportable-name as a constituent-identifier for the region that is the interface denoted by the identifier of the export-part that contains the export-clause. The occurrence of an identifier in an export-renaming-clause of an export-clause shall constitute the defining-point of that identifier as a constituent-identifier for the region that is the interface denoted by the identifier of the export-part that contains the export-clause.

A constituent-identifier so defined shall denote: the value denoted by the constant-name; the type, bindability, and initial state denoted by the type-name; the schema denoted by the schema-name; the variable denoted by the variable-name; the procedure denoted by the procedure-name; or the function denoted by the function-name; that is contained by the export-clause. That constituent-identifier shall be designated protected (see **6.5.1**) if the export-clause contains either protected or a protected variable-identifier. The type possessed by a protected constituent-identifier shall be protectable. The constituent-identifier shall be designated a module-parameter if and only if the export-clause contains a variable-identifier that is a module-parameter. If the constituent-identifier denotes a value, it shall be designated a principal identifier (see **6.4.2.3**) of that value if the constant-identifier contained by the export-clause is a principal identifier of the value and the export-clause does not contain an export-renaming-clause.

NOTE — 1 A principal identifier of a value is exported as a principal identifier only if it is not renamed. Renaming a principal identifier exports a new identifier for the value, but the new identifier is not a principal identifier.

The constant-names of the first-constant-name and of the last-constant-name of an export-range shall denote values of the same type, which shall be an enumerated-type; these values and type shall be designated the *least-value*, *greatest-value*, and *type* of the export-range, respectively. The least-value shall not exceed the greatest-value.

For each value of the type of an export-range not smaller than the least-value of the export-range and not larger than the greatest-value of the export-range

a) the export-range shall be within the scope of a defining-point of an identifier that is a principal identifier of the value;

b) the occurrence of the export-range shall constitute the defining-point of that identifier as a constituent-identifier for the region that is the interface denoted by the identifier of the export-part that contains the export-range; and

c) the constituent-identifier so defined shall denote that value and shall be designated a principal identifier of that value.

NOTES

2 Only the identifiers specified in an export-list are exported. In particular, the constant-identifiers of an enumerated-type are not exported by exporting the type-identifier.

3 Although the field-identifiers of a record-type cannot be exported, they are available in any block that can access a variable, constant, or function result possessing the record-type.

4 Although the discriminant-identifiers of a schema cannot be exported, they are available in any block that can access a variable or constant possessing a type produced from the schema.

5 Protected variable-names excepted, a constant-name, type-name, schema-name, variable-name, procedure-name, or function-name that is passed through an interface by a constituent-identifier behaves the same as a constant-name, type-name, schema-name, variable-name, procedure-name, or function-name that does not pass through an interface.

6 An export-range serves to export only the principal identifiers of the values within the specified range; it is essentially a shorthand notation for listing the principal identifiers for each value. The names that are specified in the export-range serve only to denote the least and greatest values and are not themselves exported unless they happen to be the principal identifiers of those values.

The required interface-identifiers and required constituent-identifiers shall be as specified in **6.11.4.2**.

6.11.3 Import-specifications

An import-specification shall introduce an identifier to denote an interface and zero or more identifiers, each of which shall be designated *imported*.

import-specification = interface-identifier [access-qualifier] [import-qualifier] .

access-qualifier = 'qualified' .

import-qualifier = [selective-import-option] '(' import-list ')' .

selective-import-option = 'only' .

import-list = import-clause { ',' import-clause } .

import-clause = constituent-identifier | import-renaming-clause .

import-renaming-clause = constituent-identifier '=>' identifier .

imported-interface-identifier = identifier .

Each imported identifier shall be said to *correspond* to a constituent-identifier of the interface.

For each constituent-identifier having a defining-point for the region that is the interface denoted by the interface-identifier of an import-specification

a) the occurrence of that interface-identifier shall constitute the defining-point of that constituent-identifier for each region that is a constituent-identifier contained by the import-specification.

b) for each applied occurrence of the constituent-identifier in an import-clause contained by the import-specification, a distinct imported identifier shall be introduced with the import-clause as its defining-point and with the spelling of the constituent-identifier. If that constituent-identifier is a principal identifier of a value, the imported identifier shall be designated a principal identifier of that value.

c) for each applied occurrence of the constituent-identifier in an import-renaming-clause of an import-clause contained by the import-specification, a distinct imported identifier shall be introduced with the import-clause as its defining-point and with the spelling of the identifier of the import-renaming-clause.

d) if the import-specification does not contain a selective-import-option, then for each constituent-identifier that does not have an applied occurrence contained by the import-specification, a distinct imported identifier shall be introduced with the import-specification as its defining-point and with the spelling of the constituent-identifier. If that constituent-identifier is a principal identifier of a value, the imported identifier shall be designated a principal identifier of that value.

NOTE — 1 A principal identifier of a value is imported as a principal identifier only if it is not renamed. Renaming a principal identifier imports a new identifier for the value, but the new identifier is not a principal identifier.

An imported identifier corresponding to a constituent-identifier shall be: a constant-identifier that denotes the value; a type-identifier that denotes the type, bindability, and initial state; a schema-identifier that denotes the schema; a variable-identifier that denotes the variable; a procedure-identifier that denotes the procedure; or a function-identifier that denotes the function; denoted by the constituent-identifier. An imported variable-identifier corresponding to a protected constituent-identifier shall be designated *protected*. An imported variable-identifier shall be designated a module-parameter if and only if it corresponds to a constituent-identifier that is a module-parameter.

The occurrence of an interface-identifier in an import-specification shall constitute the defining-point of the identifier of the interface-identifier as an imported-interface-identifier for the region that is the block, module-heading, or module-block closest-containing the import-specification. Each imported identifier in the set of imported identifiers determined by the import-specification shall be said to be *associated* with that imported-interface-identifier.

Each defining-point of an imported identifier occurring within an import-specification shall be for the region that is the import-specification, and, if an access-qualifier does not occur in the import-specification, also for the region that is the module-heading, module-block, or block closest-containing the import-specification.

NOTE — 2 If the access-qualifier qualified does occur in the import-specification, then imported identifiers can only be referred to within the module-heading, module-block, or containing block by their full name, which includes the interface-identifier.

6.11.4 Required interfaces

6.11.4.1 General

The required interface-identifiers and constituent-identifiers shall be defined as follows.

6.11.4.2 StandardInput and StandardOutput

The required interface-identifier **StandardInput** shall denote the required interface composed of the required constituent-identifier **input**. The constituent-identifier shall denote the required textfile **input**.

The required interface-identifier **StandardOutput** shall denote the required interface composed of the required constituent-identifier **output**. The constituent-identifier shall denote the required textfile **output**.

The required textfile **input** or **output** shall be designated *implicitly accessible* by a procedure-statement or a function-designator if and only if one or more of the following five conditions is true.

a) The procedure-statement or function-designator is contained by a block, module-heading, or module-block closest-containing an applied occurrence of the required identifier **StandardInput** or **StandardOutput**, respectively.

b) The procedure-statement or function-designator is contained by a module-block, and the module-heading associated with the module-block contains an applied occurrence of the required identifier **StandardInput** or **StandardOutput**, respectively.

c) The procedure-statement or function-designator is contained by the main-program-declaration, which contains a program-parameter-list containing the identifier **input** or **output**, respectively.

d) The procedure-statement or function-designator is contained by a module-heading or its associated module-block, and the module-parameter-list of the module-heading contains the identifier **input** or **output**, respectively.

e) The block is contained by a module-block, and the associated module-heading contains a module-parameter-list containing the identifier **input** or **output**, respectively.

The activation of the program-block of a program containing a block or module-block in which the required textfile **input** is implicitly accessible shall cause the post-assertions of **reset** to hold prior to the first access to the textfile or its associated buffer-variable. The effect of the application of the required procedures **reset**, **rewrite**, or **extend** to the textfile shall be implementation-defined.

The activation of the program-block of a program containing a block or module-block in which the required textfile **output** is implicitly accessible shall cause 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 procedures **reset**, **rewrite**, or **extend** to the textfile shall be implementation-defined.

6.11.5 Example of a module

```
module  RandomUniform  interface;
{ RandomUniform provides the pseudo-random number generator based on the
one designed by Wichmann and Hill, as described in their note 'Building
a Random-Number Generator', Byte, March 1987, pp.127-128
}
  export
    RandomUniform = (random, setseed, getseed, seedtype,
                           seedmin, seedmax, seedinit);

  const
    p1 = 30269;  m1 = 171;
    p2 = 30307;  m2 = 172;
    p3 = 30323;  m3 = 170;
  type
    seedtype = record
                 s1: 1..p1-1;
                 s2: 1..p2-1;
                 s3: 1..p3-1
               end;
  const
    seedmin  = seedtype[s1,s2,s3:1];
    seedmax  = seedtype[s1:p1-1; s2:p2-1; s3:p3-1];
    seedinit = seedtype[s1:1; s2:10000; s3:3000];

  procedure
```

```
    setseed (s:seedtype);
  procedure
    getseed (var s:seedtype);
  function
    random: real;

end.  { of RandomUniform heading }

module  RandomUniform  implementation;
{ An implementation of RandomUniform that assumes
  maxint >= largestof(p1,p2,p3)  (= 30323)
 }

  var
    seed: seedtype value seedinit;

  procedure setseed;
  begin
    seed := s
  end;

  procedure getseed;
  begin
    s := seed
  end;

  function random;
    var
      x1,x2,x3: integer;
      temp: real;
  begin
    with seed do
    begin
      { first generator }
      x1 := m1*(s1 mod 177) - 2*(s1 div 177);
      if x1<0 then x1 := x1+p1;
      { second generator }
      x2 := m2*(s2 mod 176) - 35*(s2 div 176);
      if x2<0 then x2 := x2+p2;
      { third generator }
      x3 := m3*(s3 mod 178) - 63*(s3 div 178);
      if x3<0 then x3 := x3+p3;
      { form new seed and function result }
      seed := seedtype[s1:x1; s2:x2; s3:x3];
      temp := s1/p1 + s2/p2 + s3/p3;
      random := temp-trunc(temp)
```

```
    end
  end;  { of random }

end.  { of RandomUniform block }

module  RandomUniform  implementation;
{ An alternative implementation of RandomUniform that assumes
  maxint >= largestof((p1-1)*m1,(p2-1)*m2,(p3-1)*m3)   (= 5212632)
  by using larger integers, this will run faster on many machines
}

  var
    seed: seedtype value seedinit;

  procedure setseed;
  begin
    seed := s
  end;

  procedure getseed;
  begin
    s := seed
  end;

  function random;
    var
      temp: real;
  begin
    with seed do
    begin
      { form new seed }
      s1 := (m1*s1) mod p1;   { first generator }
      s2 := (m2*s2) mod p2;   { second generator }
      s3 := (m3*s3) mod p3;   { third generator }
      { form function result }
      temp := s1/p1 + s2/p2 + s3/p3;
      random := temp-trunc(temp)
    end
  end;  { of random }

end.  { of RandomUniform block }
```

6.11.6 Examples of program-components that are module-declarations

NOTE — 1 Each of examples 2 to 5 depends on one or more of examples 1 to 4 that precede it.

Example 1:

```
module m1;
{ m1 exports one interface named i1, containing two values named low and high.
The variable null is not exported.  m1 has a minimal module-block.}

  export i1 = (low,high);

  const  low = 0;  high = 1;

  var    null: record end;

end { of module-heading for m1 } ;

end { of module-block for m1 } .
```

Example 2:

```
module m2;
{ m2 exports two interfaces named i2 and j2.  i2 contains a type called t;  j2
contains the two values (still named low and high) imported from m1 through
interface i1.  m2 also has a minimal module-block.}

  export
    i2 = (t);  { define i2 to have t as its only constituent-identifiers. }
    j2 = (low,high); { re-export low and high in j2.  They are
                    imported through interface i1. }
  import
    i1;     { import all constituent-identifiers of i1 }

  type t = low..high;

end { of module-heading for m2 } ;

end { of module-block for m2 } .
```

Example 3:

```
module m3;
```

```
{ m3 exports one interface containing a function, a type, and two values.  The
function-heading is declared in the module-heading, and the function-block is
declared in the module-block. }

  export
    i3 = (f, i2.f_range, i1.low=>f_low, i1.high=>f_high);
      { Export constituent-identifiers f, f_range, f_low, and
        f_high. }

  import
    i1 qualified;  { Import all constituent-identifiers from i1.  Within this
                     module they are named i1.low and i1.high. }
    i2 qualified only (t=>f_range);
                   { Import only t through i2.  Within this
                     module it is named i2.f_range. }

  function f(x: integer): i2.f_range;

end { of module-heading for m3 } ;

  function f;
  begin
    if x < i1.low then f := i1.low
    else if x > i1.high then f := i1.high
    else f := x
  end { f } ;

end { of module-block for m3 } .
```

Example 4:

```
module m4 interface;
{ m4 exports two interfaces named enq and deq.  enq contains a
procedure named enqueue.  deq contains a procedure called dequeue, a
function called empty, and a type called range.  The module-block is given
separately in example 5.}

  export enq = (enqueue); deq = (dequeue,empty,range);

  import i3 only (f_range => range);
    { Import only f_range through i3.  Within m4 it is named
      range. }
```

```
  procedure enqueue(e: range);
  procedure dequeue(var e: range);
  function empty: Boolean;

end { of module-heading for m4 } .
```

Example 5:

```
module m4 implementation;
{ This is the module-block of m4. Note that any other program-components could
be placed between examples 4 and 5. All identifiers and interfaces that are
visible in the module-heading 4 are also visible here. }

  type
    qp = ↑ qnode;
    qnode = record next: qp; c: range end;

  var
    oldest: qp value nil; { initialize queue to empty }
    newest: qp;

  function empty;
  begin  empty := (oldest = nil) end { empty } ;

  procedure enqueue;
  begin
    if empty then
      begin new(newest); oldest := newest end
    else
      begin new(newest↑.next); newest := newest↑.next end;
    newest↑.c := e
  end { enqueue } ;

  procedure dequeue;
    var p: qp;
  begin
    if empty then halt;
    e := oldest↑.c;  p := oldest;
    if oldest = newest then oldest := nil
    else oldest := oldest↑.next;
    dispose(p)
  end { dequeue } ;
```

```
end { of module-block for m4 } .
```

NOTE — 2 Each of examples 7 to 9 depends on one or more of examples 6 to 8 that precede it.

Example 6:

```
module generic_sort interface;

   export generic_sort = (do_the_sort,max_sort_index,
                   protected current_pass => number_of_passes,
                   protected swap_occurred_during_sort);

   { the export of current_pass and swap_occurred_during_sort allows the }
   {    caller of the sort procedure to determine the status of the sort }
   {    --- they are marked as protected so that the caller is not       }
   {    allowed write access                                             }

   { current_pass is renamed to number_of_passes as it is exported  }

   type max_sort_index = 1..maxint;

   procedure do_the_sort(element_count : max_sort_index;
               function greater(e1,e2 : max_sort_index) :Boolean;
               procedure swap(e1,e2 : max_sort_index) );

   var current_pass : 0..maxint value 0;
       swap_occurred_during_sort : Boolean value false;

end.
```

Example 7:

```
module generic_sort implementation;

   procedure do_the_sort;

   var swap_occurred_this_pass : Boolean;
       n : max_sort_index;
   begin
     current_pass := 0;
     swap_occurred_during_sort := false;
```

```
    repeat
       swap_occurred_this_pass := false;
       current_pass := current_pass + 1;
       for n := 1 to element_count - 1 do
          if greater(n,n + 1) then begin
             swap(n,n + 1);
             swap_occurred_this_pass := true;
             end;
       swap_occurred_during_sort := swap_occurred_during_sort or
                      swap_occurred_this_pass;
    until not swap_occurred_this_pass;
  end;

end.
```

Example 8:

```
module employee_sort interface;

  export employee_sort = (sort_by_name,sort_by_clock_number,employee_list);

  import generic_sort;

  type
     employee = record
        last_name,first_name : string(30);
        clock_number : 1..maxint;
     end;

     employee_list(num_employees : max_sort_index) =
        array [1..num_employees] of employee;

  procedure sort_by_name(employees : employee_list;
               var something_done : Boolean);

  procedure sort_by_clock_number(employees : employee_list;
               var something_done : Boolean);

end.
```

Example 9:

```
module employee_sort implementation;

   procedure sort_by_name;

     procedure swap_employees(e1,e2 : max_sort_index);

        var temp : employee;

     begin
        temp := employees[e1];
        employees[e2] := employees[e1];
        employees[e1] := temp;
     end;

     function name_is_greater(e1,e2 : max_sort_index);

     begin
       name_is_greater := (employees[e1].last_name > employees[e2].last_name)
              or ( (employees[e1].last_name = employees[e2].last_name) and
                 (employees[e1].first_name > employees[e2].first_name));
     end;

   begin { sort_by_name }
      do_the_sort(employees.num_employees,name_is_greater,swap_employees );
      something_done := swap_occurred_during_sort;
   end; { sort_by_name }

   procedure sort_by_clock_number;

     procedure swap_employees(e1,e2 : max_sort_index);

       var temp : employee;

     begin
       temp := employees[e1];
       employees[e2] := employees[e1];
       employees[e1] := temp;
     end;

     function clock_is_greater(e1,e2 : max_sort_index);

     begin
       clock_is_greater := employees[e1].clock_number >
               employees[e2].clock_number;
```

```
     end;

   begin { sort_by_clock_number }
     do_the_sort(employees.num_employees,clock_is_greater,swap_employees );
     something_done := swap_occurred_during_sort;
   end; { sort_by_clock_number }

end.
```

Example 10:

```
module event_recording ;

   export event_recording = (record_event);

   { This procedure allows recording of user-specified events to }
   {    a log file.   The event to be recorded is specified as a  }
   {    string, and a time stamp is added automatically.         }

   procedure record_event(event_to_record : string);

end;

var
   logfile : bindable text;
   logbind : BindingType;

procedure record_event;

var stamp : TimeStamp;

begin
GetTimeStamp(stamp);
writeln(logfile,'event ',event_to_record,' occurred on ',
          date(stamp),' at ',time(stamp));
end;

{ this module needs an initialization section to open the log file }
{    and a termination section to close it                         }

to begin do
   begin
   logbind := Binding(logfile);
```

```
   logbind.name := 'logfile';
   bind(logfile,logbind);
   rewrite(logfile);
   record_event('event-module initialization');
   end;

to end do
   begin
   record_event('event-module termination');
   unbind(logfile);
   end;

end.
```

6.11.7 Example of exporting a range of enumerated-type values

```
module line_parameters interface;

   export
     line_states = (line_state, onhook..permanent_sequence,
         first_state, last_state, initial_state);
        { 'onhook..permanent_sequence' exports each value name
          defined in the type definition. }

      type
         line_state = (onhook, offhook, tone_applied,
           ringing_applied, two_way_talk, out_of_service,
           permanent_sequence);

      const
           first_state = onhook;
           last_state = permanent_sequence;
           initial_state = out_of_service;

end.

module line_interfaces interface;

  export
     standard_line_interface = (first_state..last_state,
         first_state, last_state, initial_state, set_state,
         current_state);
      { 'first_state..last_state' exports 'onhook' through
        'permanent_sequence', but not 'first_state' and
        'last_state'; 'first_state, last_state' exports
        'first_state' and 'last_state'. }
```

```
    import
      line_states;

      procedure set_state(new_state: line_state);

      function current_state: line_state;

end.
```

6.12 Main-program-declarations

main-program-declaration = program-heading ';' main-program-block .

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

program-parameter-list = identifier-list .

main-program-block = block .

The identifiers contained by the program-parameter-list of a program-heading of a main-program-declaration shall have distinct spellings, and for each such identifier there shall be a defining-point as a variable-identifier with the same spelling for the region that is the block of the main-program-block of the main-program-declaration. If the spelling is neither **input** nor **output**, the variable-identifier either shall be local to the block or shall be an imported variable-identifier that is a module-parameter. If the spelling is **input** or **output**, the occurrence of the identifier contained by the program-parameter-list shall constitute a defining-point for the region that is the block of the main-program-block as a variable-identifier denoting the required textfile **input** or **output**, respectively. If the variable-identifier is local to the block or has the spelling **input** or **output**, both the variable-identifier and any variable it denotes shall be designated a *program-parameter*. The binding of a variable that is a program-parameter to entities external to the program shall be implementation-defined.

NOTES

1 The external representation of such external entities is not defined by this International Standard.

2 Variables that are program-parameters are not necessarily bound when the program is activated.

3 See **6.11.4.2** regarding **reset**, **rewrite**, and **extend** for the required textfiles **input** and **output**.

Examples:

```
1)  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.
```

2)

```
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.
```

3)

```
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 }
```

4)

```
program clear_my_screens;

    type

    positive = 1..maxint;

    graphic_screen(max_rows,max_cols,bits_per_pixel : positive) =
        packed array [0..max_rows-1,0..max_cols-1] of
           set of 0..bits_per_pixel-1 ;

    var

    medium_res_mono : graphic_screen(512,512,1);
    highres_mono : graphic_screen(1024,1024,1);
    lowres_color : graphic_screen(256,256,3); { 8 colors }
    super_highres_technicolor : graphic_screen(4096,4096,16);
                                { 65536 colors }

    procedure clear_screen(var scr : graphic_screen);

    var m,n : 0 .. maxint - 1;
```

```
begin
for n := 0 to scr.max_rows - 1 do
   for m := 0 to scr.max_cols - 1 do
      scr[n,m] := [];
end;

begin { main program }
clear_screen(medium_res_mono);
clear_screen(highres_mono);
clear_screen(lowres_color);
clear_screen(super_highres_technicolor);
end. { main program }
```

6.13 Programs

program = program-block .

program-block = program-component { program-component } .

program-component = main-program-declaration '.' | module-declaration '.' .

A program-block shall contain exactly one main-program-declaration.

A processor should be able to accept the program-components of the program-block separately.

NOTES

1 This International Standard constrains the order of program-components of a conforming program only by the partial ordering defined by **6.2.2.9**. A further restriction by a processor on the order of program-components can be justified only by subclause **1.2 a)**.

2 This International Standard does not contain mechanisms for interfacing with other languages. If such a facility is implemented as an extension, it is recommended that a processor enforce the requirements of Extended Pascal pertaining to type compatibility. This facility could be provided in one of the following ways.

a) The use of module-parameters and program-parameters to denote variables, procedures, and other entities that the processor can handle. This would require some extensions, e.g., that a procedure not contain a block if it is a module-parameter or program-parameter.

b) The extension of import-specification for the same purpose.

c) The importation of a Pascal-compatible interface that has been created by an auxiliary processor.

d) The association of a module-heading with a construct in another language that the implementation has determined to be equivalent to a module-block.

Annex A

(Informative)

Collected syntax

A.1 Production rules

The nonterminal symbols number, pointer-type, program, simple-type, simple-type-name, 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 of the grammar. The subclause of definition appears at the left of each production.

6.11.3 access-qualifier = ‘qualified’ .

6.4.8 actual-discriminant-part = ‘(’ discriminant-value { ‘,’ discriminant-value } ‘)’ .

6.8.5 actual-parameter = expression | variable-access
| procedure-name | function-name .

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

6.8.3.1 adding-operator = ‘+’ | ‘−’ | ‘><’ | ‘or’ | ‘or_else’ .

6.1.9 apostrophe-image = ‘"’ .

6.8.8.2 array-constant = constant-access .

6.8.6.2 array-function = function-access .

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

6.4.1 array-type-name = type-name .

6.8.7.2 array-value = ‘[’ [array-value-element { ‘;’ array-value-element } [‘;’]]
[array-value-completer [‘;’]] ‘]’ .

6.8.7.2 array-value-completer = ‘otherwise’ component-value .

6.8.7.2 array-value-element = case-constant-list ‘:’ component-value .

6.5.3.2 array-variable = variable-access .

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

6.4.3.5 base-type = ordinal-type .

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

6.8.3.3 Boolean-expression = expression .

6.7.3.7.1 bound-identifier = identifier .

6.5.5 buffer-variable = file-variable '↑' .

6.4.3.4 case-constant = constant-expression .

6.4.3.4 case-constant-list = case-range { ',' case-range } .

6.9.3.5 case-index = expression .

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

6.4.3.4 case-range = case-constant ['..' case-constant] .

6.9.3.5 case-statement = 'case' case-index 'of'
(case-list-element { ';' case-list-element }
[[';'] case-statement-completer] | case-statement-completer)
[';'] 'end' .

6.9.3.5 case-statement-completer = 'otherwise' statement-sequence .

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

6.4.2.1 complex-type-name = type-name .

6.8.6.1 component-function-access = indexed-function-access
| record-function-access .

6.4.3.2 component-type = type-denoter .

6.8.7.1 component-value = expression | array-value | record-value .

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

6.9.3.2 compound-statement = 'begin' statement-sequence 'end' .

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

6.7.3.7.1 conformant-array-form = packed-conformant-array-form
| unpacked-conformant-array-form .

6.7.3.7.1 conformant-array-parameter-specification =
['protected'](value-conformant-array-specification
| variable-conformant-array-specification) .

6.8.8.1 constant-access = constant-access-component | constant-name .

6.8.8.1 constant-access-component = indexed-constant
| field-designated-constant
| substring-constant .

6.3.1 constant-definition = identifier '=' constant-expression .

6.2.1 constant-definition-part = 'const' constant-definition ';' { constant-definition ';' } .

6.8.2 constant-expression = expression .

6.9.3.10 constant-field-identifier = identifier .

6.3.1 constant-identifier = identifier .

6.3.1 constant-name = [imported-interface-identifier '.'] constant-identifier .

6.8.7.3 constant-tag-value = constant-expression .

6.11.2 constituent-identifier = identifier .

6.9.3.9.1 control-variable = entire-variable .

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

6.1.7 digit-sequence = digit { digit } .

6.1.4 directive = letter { [underscore] (letter | digit) } .

6.4.7 discriminant-identifier = identifier .

6.4.7 discriminant-specification = identifier-list ':' ordinal-type-name .

6.8.4 discriminant-specifier = discriminant-identifier .

6.4.8 discriminant-value = expression .

6.4.8 discriminated-schema = schema-name actual-discriminant-part .

6.4.4 domain-type = type-name | schema-name .

6.9.3.4 else-part = 'else' statement .

6.9.2.1 empty-statement = .

6.8.6.1 entire-function-access = function-designator .

6.5.2 entire-variable = variable-name .

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

6.8.3.1 exponentiating-operator = '**' | 'pow' .

6.11.2 export-clause = exportable-name | export-renaming-clause .

6.11.2 export-list = (export-clause | export-range)
{ ',' (export-clause | export-range) } .

6.11.2 export-part = identifier '=' '(' export-list ')' .

6.11.2 export-range = first-constant-name '..' last-constant-name .

6.11.2 export-renaming-clause = exportable-name '=¿' identifier .

6.11.2 exportable-name = constant-name
| type-name
| schema-name
| ['protected'] variable-name
| procedure-name
| function-name .

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

6.1.7 extended-digit = digit | letter .

6.1.7 extended-number = unsigned-integer '#' extended-digit { extended-digit } .

6.8.1 factor = primary [exponentiating-operator primary] .

6.8.8.3 field-designated-constant = record-constant '.' field-specifier
| constant-field-identifier .

6.5.3.3 field-designator = record-variable '.' field-specifier | field-designator-identifier .

6.9.3.10 field-designator-identifier = identifier .

6.4.3.4 field-identifier = identifier .

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

6.8.7.3 field-list-value = [(fixed-part-value [';' variant-part-value] | variant-part-value) [';']] .

6.5.3.3 field-specifier = field-identifier .

6.8.7.3 field-value = field-identifier { ',' field-identifier }
':' component-value .

6.4.3.6 file-type = 'file' ['[' index-type ']'] 'of' component-type .

6.4.1 file-type-name = type-name .

6.5.5 file-variable = variable-access .

6.9.3.9.2 final-value = expression .

6.11.1 finalization-part = 'to' 'end' 'do' statement ';' .

6.11.2 first-constant-name = constant-name .

6.4.3.4 fixed-part = record-section { ';' record-section } .

6.8.7.3 fixed-part-value = field-value { ';' field-value } .

6.9.3.9.1 for-statement = 'for' control-variable iteration-clause 'do' statement .

6.4.7 formal-discriminant-part = '(' discriminant-specification
{ ';' discriminant-specification } ')' .

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

6.7.3.1 formal-parameter-section > value-parameter-specification
| variable-parameter-specification
| procedural-parameter-specification
| functional-parameter-specification .

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

6.1.7 fractional-part = digit-sequence .

6.8.6.1 function-access = entire-function-access
| component-function-access
| substring-function-access .

6.7.2 function-block = block .

6.7.2 function-declaration = function-heading ';' remote-directive
| function-identification ';' function-block
| function-heading ';' function-block .

6.8.5 function-designator = function-name [actual-parameter-list] .

6.7.2 function-heading = 'function' identifier [formal-parameter-list]
[result-variable-specification] ':' result-type .

6.7.2 function-identification = 'function' function-identifier .

6.8.6.4 function-identified-variable = pointer-function '↑' .

6.7.2 function-identifier = identifier .

6.7.2 function-name = [imported-interface-identifier '.'] function-identifier .

6.7.3.1 functional-parameter-specification = function-heading .

6.9.2.4 goto-statement = ‘goto’ label .

6.5.4 identified-variable = pointer-variable ‘↑’ .

6.1.3 identifier = letter { [underscore] (letter | digit) } .

6.4.2.3 identifier-list = identifier { ‘,’ identifier } .

6.9.3.4 if-statement = ‘if’ Boolean-expression ‘then’ statement [else-part] .

6.1.6 implementation-directive = directive .

6.11.3 import-clause = constituent-identifier | import-renaming-clause .

6.11.3 import-list = import-clause { ‘,’ import-clause } .

6.2.1 import-part = [‘import’ import-specification ‘;’ { import-specification ‘;’ }] .

6.11.3 import-qualifier = [selective-import-option] ‘(’ import-list ‘)’ .

6.11.3 import-renaming-clause = constituent-identifier ‘=¿’ identifier .

6.11.3 import-specification = interface-identifier [access-qualifier]
[import-qualifier] .

6.11.3 imported-interface-identifier = identifier .

6.5.3.2 index-expression = expression .

6.4.3.2 index-type = ordinal-type .

6.7.3.7.1 index-type-specification = identifier ‘..’ identifier ‘:’ ordinal-type-name .

6.8.8.2 indexed-constant = array-constant ‘[’ index-expression { ‘,’ index-expression } ‘]’
| string-constant ‘[’ index-expression ‘]’ .

6.8.6.2 indexed-function-access = array-function ‘[’ index-expression { ‘,’ index-expression } ‘]’
| string-function ‘[’ index-expression ‘]’ .

6.5.3.2 indexed-variable = array-variable ‘[’ index-expression { ‘,’ index-expression } ‘]’
| string-variable ‘[’ index-expression ‘]’ .

6.6 initial-state-specifier = ‘value’ component-value .

6.9.3.9.2 initial-value = expression .

6.11.1 initialization-part = ‘to’ ‘begin’ ‘do’ statement ‘;’ .

6.1.5 interface-directive = directive .

6.11.2 interface-identifier = identifier .

6.11.2 interface-specification-part = ‘export’ export-part ‘;’ { export-part ‘;’ } .

6.9.3.9.1 iteration-clause = sequence-iteration | set-member-iteration .

6.1.8 label = digit-sequence .

6.2.1 label-declaration-part = 'label' label { ',' label } ';' .

6.11.2 last-constant-name = constant-name .

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' .

6.12 main-program-block = block .

6.12 main-program-declaration = program-heading ';' main-program-block .

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

6.11.1 module-block = import-part
{ constant-definition-part
| type-definition-part
| variable-declaration-part
| procedure-and-function-declaration-part }
[initialization-part]
[finalization-part]
'end' .

6.11.1 module-declaration = module-heading [';' module-block]
| module-identification ';' module-block .

6.11.1 module-heading = 'module' identifier [interface-directive]
['(' module-parameter-list ')'] ';'
interface-specification-part
import-part
{ constant-definition-part
| type-definition-part
| variable-declaration-part
| procedure-and-function-heading-part }
'end' .

6.11.1 module-identification = 'module' module-identifier implementation-directive .

6.11.1 module-identifier = identifier .

6.11.1 module-parameter-list = identifier-list .

6.8.3.1 multiplying-operator = '*' | '/' | 'div' | 'mod' | 'and' | 'and_then' .

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
| restricted-type .

6.1.7 number = signed-number
| [sign] (digit-sequence '.' | '.' fractional-part) ['e' scale-factor] .

6.4.2.1 ordinal-type = new-ordinal-type | ordinal-type-name
| type-inquiry | discriminated-schema .

6.4.2.1 ordinal-type-name = type-name .

6.7.3.7.1 packed-conformant-array-form = 'packed' 'array' '[' index-type-specification ']' 'of' type-name .

6.7.3.1 parameter-form = type-name | schema-name | type-inquiry .

6.7.3.1 parameter-identifier = identifier .

6.8.6.4 pointer-function = function-access .

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

6.4.1 pointer-type-name = type-name .

6.5.4 pointer-variable = variable-access .

6.8.1 primary > variable-access | unsigned-constant | set-constructor
| function-access | '(' expression ')' | 'not' primary
| constant-access | schema-discriminant
| structured-value-constructor | discriminant-identifier .

6.7.3.7.1 primary > bound-identifier .

6.7.3.1 procedural-parameter-specification = procedure-heading .

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

6.11.1 procedure-and-function-heading-part = (procedure-heading | function-heading) ';' .

6.7.1 procedure-block = block .

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

6.7.1 procedure-heading = ‘procedure’ identifier [formal-parameter-list] .

6.7.1 procedure-identification = ‘procedure’ procedure-identifier .

6.7.1 procedure-identifier = identifier .

6.7.1 procedure-name = [imported-interface-identifier ‘.’] procedure-identifier .

6.9.2.3 procedure-statement = procedure-name ([actual-parameter-list]
| read-parameter-list | readln-parameter-list | readstr-parameter-list
| write-parameter-list | writeln-parameter-list | writestr-parameter-list) .

6.13 program = program-block .

6.13 program-block = program-component { program-component } .

6.13 program-component = main-program-declaration ‘.’ | module-declaration ‘.’ .

6.12 program-heading = ‘program’ identifier [‘(’ program-parameter-list ‘)’] .

6.12 program-parameter-list = identifier-list .

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

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

6.7.5.5 readstr-parameter-list = ‘(’ string-expression ‘,’ variable-access { ‘,’ variable-access } ‘)’ .

6.4.2.1 real-type-name = type-name .

6.8.8.3 record-constant = constant-access .

6.8.6.3 record-function = function-access .

6.8.6.3 record-function-access = record-function ‘.’ field-specifier .

6.4.3.4 record-section = identifier-list ‘:’ type-denoter .

6.4.3.4 record-type = ‘record’ field-list ‘end’ .

6.4.1 record-type-name = type-name .

6.8.7.3 record-value = ‘[’ field-list-value ‘]’ .

6.5.3.3 record-variable = variable-access .

6.8.3.1 relational-operator = ‘=’ | ‘<>’ | ‘<’ | ‘>’ | ‘<=’ | ‘>=’ | ‘in’ .

6.1.4 remote-directive = directive .

6.9.3.7 repeat-statement = ‘repeat’ statement-sequence ‘until’ Boolean-expression .

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

6.4.2.5 restricted-type = 'restricted' type-name .

6.7.2 result-type = type-name .

6.7.2 result-variable-specification = '=' identifier .

6.1.7 scale-factor = [sign] digit-sequence .

6.4.7 schema-definition = identifier '=' schema-name
| identifier formal-discriminant-part '=' type-denoter .

6.8.4 schema-discriminant = (variable-access | constant-access) '.' discriminant-specifier
| schema-discriminant-identifier .

6.9.3.10 schema-discriminant-identifier = identifier .

6.4.7 schema-identifier = identifier .

6.4.7 schema-name = [imported-interface-identifier '.'] schema-identifier .

6.11.3 selective-import-option = 'only' .

6.9.3.9.2 sequence-iteration = ':=' initial-value ('to' | 'downto') final-value .

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

6.9.3.9.3 set-expression = expression .

6.9.3.9.3 set-member-iteration = 'in' set-expression .

6.4.3.5 set-type = 'set' 'of' base-type .

6.4.1 set-type-name = type-name .

6.8.7.4 set-value = set-constructor .

6.1.7 sign = '+' | '−' .

6.1.7 signed-integer = [sign] unsigned-integer .

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

6.1.7 signed-real = [sign] unsigned-real .

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

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

6.4.2.1 simple-type = ordinal-type | real-type-name | complex-type-name .

6.4.1 simple-type-name = type-name .

6.1.2 special-symbol = ‘+’ | ‘−’ | ‘*’ | ‘/’ | ‘=’ | ‘<’ | ‘>’ | ‘[’ | ‘]’
| ‘.’ | ‘,’ | ‘:’ | ‘;’ | ‘↑’ | ‘(’ | ‘)’ | ‘**’
| ‘<>’ | ‘<=’ | ‘>=’ | ‘:=’ | ‘..’ | ‘><’ | ‘=>’
| word-symbol .

6.9.1 statement = [label ‘:’] (simple-statement | structured-statement) .

6.2.1 statement-part = compound-statement .

6.9.3.1 statement-sequence = statement { ‘;’ statement } .

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

6.8.8.2 string-constant = constant-access .

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

6.7.5.5 string-expression = expression .

6.8.6.2 string-function = function-access .

6.5.3.2 string-variable = variable-access .

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

6.4.3.1 structured-type = new-structured-type | structured-type-name .

6.4.1 structured-type-name = array-type-name
| record-type-name
| set-type-name
| file-type-name .

6.8.7.1 structured-value-constructor = array-type-name array-value
| record-type-name record-value
| set-type-name set-value .

6.4.2.4 subrange-bound = expression .

6.4.2.4 subrange-type = subrange-bound ‘..’ subrange-bound .

6.8.8.4 substring-constant = string-constant ‘[’ index-expression ‘..’ index-expression ‘]’ .

6.8.6.5 substring-function-access = string-function ‘[’ index-expression ‘..’ index-expression ‘]’ .

6.5.6 substring-variable = string-variable ‘[’ index-expression ‘..’ index-expression ‘]’ .

6.4.3.4 tag-field = identifier .

6.8.7.3 tag-field-identifier = field-identifier .

6.4.3.4 tag-type = ordinal-type-name .

6.8.1 term = factor { multiplying-operator factor } .

6.4.1 type-definition = identifier '=' type-denoter .

6.2.1 type-definition-part = 'type' (type-definition | schema-definition) ';'
{ (type-definition | schema-definition) ';' } .

6.4.1 type-denoter = ['bindable'] (type-name | new-type
| type-inquiry | discriminated-schema)
[initial-state-specifier] .

6.4.1 type-identifier = identifier .

6.4.9 type-inquiry = 'type' 'of' type-inquiry-object .

6.4.9 type-inquiry-object = variable-name | parameter-identifier .

6.4.1 type-name = [imported-interface-identifier '.'] type-identifier .

6.1.3 underscore = '_' .

6.7.3.7.1 unpacked-conformant-array-form =
'array' '[' index-type-specification { ';' index-type-specification } ']'
'of' (type-name | conformant-array-form) .

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

6.8.1 unsigned-constant = unsigned-number | character-string | 'nil' | extended-number .

6.1.7 unsigned-integer = digit-sequence .

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

6.1.7 unsigned-real = digit-sequence '.' fractional-part ['e' scale-factor]
| digit-sequence 'e' scale-factor .

6.7.3.7.1 value-conformant-array-specification = identifier-list ':' conformant-array-form .

6.7.3.1 value-parameter-specification = ['protected'] identifier-list ':' parameter-form .

6.5.1 variable-access = entire-variable | component-variable
| identified-variable | buffer-variable
| substring-variable | function-identified-variable .

6.7.3.7.1 variable-conformant-array-specification = 'var' identifier-list ':' conformant-array-form .

6.5.1 variable-declaration = identifier-list ':' type-denoter .

6.2.1 variable-declaration-part = 'var' variable-declaration ';' { variable-declaration ';' } .

6.5.1 variable-identifier = identifier .

6.5.1 variable-name = [imported-interface-identifier '.'] variable-identifier .

6.7.3.1 variable-parameter-specification = ['protected'] 'var' identifier-list ':' parameter-form .

6.4.3.4 variant-denoter = '(' field-list ')' .

6.4.3.4 variant-list-element = case-constant-list ':' variant-denoter .

6.4.3.4 variant-part = 'case' variant-selector 'of'
(variant-list-element { ';' variant-list-element }
[[';'] variant-part-completer]
| variant-part-completer) .

6.4.3.4 variant-part-completer = 'otherwise' variant-denoter .

6.8.7.3 variant-part-value = 'case' [tag-field-identifier ':']
constant-tag-value 'of' '[' field-list-value ']' .

6.4.3.4 variant-selector = [tag-field ':'] tag-type | discriminant-identifier .

6.9.3.8 while-statement = 'while' Boolean-expression 'do' statement .

6.9.3.10 with-element = variable-access | constant-access .

6.9.3.10 with-list = with-element { ',' with-element } .

6.9.3.10 with-statement = 'with' with-list 'do' statement .

6.1.2 word-symbol = 'and' | 'and_then' | 'array' | 'begin' | 'bindable' | 'case'
| 'const' | 'div' | 'do' | 'downto' | 'else' | 'end' | 'export'
| 'file' | 'for' | 'function' | 'goto' | 'if' | 'import'
| 'in' | 'label' | 'mod' | 'module' | 'nil' | 'not' | 'of'
| 'only' | 'or' | 'or_else' | 'otherwise' | 'packed' | 'pow'
| 'procedure' | 'program' | 'protected' | 'qualified'
| 'record' | 'repeat' | 'restricted' | 'set' | 'then' | 'to'
| 'type' | 'until' | 'value' | 'var' | 'while' | 'with' .

6.10.3 write-parameter = expression [':' expression [':' expression]] .

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

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

6.7.5.5 writestr-parameter-list = '(' string-variable ',' write-parameter { ',' write-parameter } ')' .

A.2 Index of terminals in A.1

```
"#" ................................ extended-number
"'" ................................ character-string
"''" ............................... apostrophe-image
"(" ................................ actual-discriminant-part
                                     actual-parameter-list
                                     enumerated-type
                                     export-part
                                     formal-discriminant-part
                                     formal-parameter-list
                                     import-qualifier
                                     module-heading
                                     primary
                                     program-heading
                                     read-parameter-list
                                     readln-parameter-list
                                     readstr-parameter-list
                                     special-symbol
                                     variant-denoter
                                     write-parameter-list
                                     writeln-parameter-list
                                     writestr-parameter-list
")" ................................ actual-discriminant-part
                                     actual-parameter-list
                                     enumerated-type
                                     export-part
                                     formal-discriminant-part
                                     formal-parameter-list
                                     import-qualifier
                                     module-heading
                                     primary
                                     program-heading
                                     read-parameter-list
                                     readln-parameter-list
                                     readstr-parameter-list
                                     special-symbol
                                     variant-denoter
                                     write-parameter-list
                                     writeln-parameter-list
                                     writestr-parameter-list
"*" ................................ multiplying-operator
                                     special-symbol
"**" ............................... exponentiating-operator
                                     special-symbol
"+" ................................ adding-operator
```

```
                                   record-value
                                   set-constructor
                                   special-symbol
                                   substring-constant
                                   substring-function-access
                                   substring-variable
                                   unpacked-conformant-array-form
                                   variant-part-value
"]" .............................. array-type
                                   array-value
                                   file-type
                                   indexed-constant
                                   indexed-function-access
                                   indexed-variable
                                   packed-conformant-array-form
                                   record-value
                                   set-constructor
                                   special-symbol
                                   substring-constant
                                   substring-function-access
                                   substring-variable
                                   unpacked-conformant-array-form
                                   variant-part-value
"↑" .............................. buffer-variable
                                   function-identified-variable
                                   identified-variable
                                   new-pointer-type
                                   special-symbol
"_" .............................. underscore
"a" .............................. letter
"and" ............................ multiplying-operator
                                   word-symbol
"and_then" ....................... multiplying-operator
                                   word-symbol
"array" .......................... array-type
                                   packed-conformant-array-form
                                   unpacked-conformant-array-form
                                   word-symbol
"b" .............................. letter
"begin" .......................... compound-statement
                                   initialization-part
                                   word-symbol
"bindable" ....................... type-denoter
                                   word-symbol
"c" .............................. letter
"case" ........................... case-statement
```

```
                                        word-symbol
"in" .................................. relational-operator
                                        set-member-iteration
                                        word-symbol
"j" ................................... letter
"k" ................................... letter
"l" ................................... letter
"label" ............................... label-declaration-part
                                        word-symbol
"m" ................................... letter
"mod" ................................. multiplying-operator
                                        word-symbol
"module" .............................. module-heading
                                        module-identification
                                        word-symbol
"n" ................................... letter
"nil" ................................. unsigned-constant
                                        word-symbol
"not" ................................. primary
                                        word-symbol
"o" ................................... letter
"of" .................................. array-type
                                        case-statement
                                        file-type
                                        packed-conformant-array-form
                                        set-type
                                        type-inquiry
                                        unpacked-conformant-array-form
                                        variant-part
                                        variant-part-value
                                        word-symbol
"only" ................................ selective-import-option
                                        word-symbol
"or" .................................. adding-operator
                                        word-symbol
"or_else" ............................. adding-operator
                                        word-symbol
"otherwise" ........................... array-value-completer
                                        case-statement-completer
                                        variant-part-completer
                                        word-symbol
"p" ................................... letter
"packed" .............................. new-structured-type
                                        packed-conformant-array-form
                                        word-symbol
"pow" ................................. exponentiating-operator
```

```
                                          word-symbol
"procedure" ............................. procedure-heading
                                          procedure-identification
                                          word-symbol
"program" ............................... program-heading
                                          word-symbol
"protected" ............................. conformant-array-parameter-specification
                                          exportable-name
                                          value-parameter-specification
                                          variable-parameter-specification
                                          word-symbol
"q" ..................................... letter
"qualified" ............................. access-qualifier
                                          word-symbol
"r" ..................................... letter
"record" ................................ record-type
                                          word-symbol
"repeat" ................................ repeat-statement
                                          word-symbol
"restricted" ............................ restricted-type
                                          word-symbol
"s" ..................................... letter
"set" ................................... set-type
                                          word-symbol
"t" ..................................... letter
"then" .................................. if-statement
                                          word-symbol
"to" .................................... finalization-part
                                          initialization-part
                                          sequence-iteration
                                          word-symbol
"type" .................................. type-definition-part
                                          type-inquiry
                                          word-symbol
"u" ..................................... letter
"until" ................................. repeat-statement
                                          word-symbol
"v" ..................................... letter
"value" ................................. initial-state-specifier
                                          word-symbol
"var" ................................... variable-conformant-array-specification
                                          variable-declaration-part
                                          variable-parameter-specification
                                          word-symbol
"w" ..................................... letter
"while" ................................. while-statement
```

A.3 Index of nonterminals in A.1

– a –

– b –

– c –

```
case-range ......................... case-constant-list
case-statement ..................... conditional-statement
case-statement-completer ............ case-statement
character-string ................... unsigned-constant
complex-type-name .................. simple-type
component-function-access ........... function-access
component-type ..................... array-type
                                     file-type
component-value .................... array-value-completer
                                     array-value-element
                                     field-value
                                     initial-state-specifier
component-variable ................. variable-access
compound-statement ................. statement-part
                                     structured-statement
conditional-statement ............... structured-statement
conformant-array-form ............... unpacked-conformant-array-form
                                     value-conformant-array-specification
                                     variable-conformant-array-specification
conformant-array-parameter-specification .....
                                     formal-parameter-section
constant-access .................... array-constant
                                     primary
                                     record-constant
                                     schema-discriminant
                                     string-constant
                                     with-element
constant-access-component ........... constant-access
constant-definition ................ constant-definition-part
constant-definition-part ............ block
                                     module-block
                                     module-heading
constant-expression ................ case-constant
                                     constant-definition
                                     constant-tag-value
constant-field-identifier ........... field-designated-constant
constant-identifier ................ constant-name
constant-name ...................... constant-access
                                     exportable-name
                                     first-constant-name
                                     last-constant-name
constant-tag-value ................. variant-part-value
constituent-identifier .............. import-clause
                                     import-renaming-clause
control-variable ................... for-statement
```

– d –

digit	digit-sequence
	directive
	extended-digit
	identifier
digit-sequence	fractional-part
	label
	number
	scale-factor
	unsigned-integer
	unsigned-real
directive	implementation-directive
	interface-directive
	remote-directive
discriminant-identifier	discriminant-specifier
	primary
	variant-selector
discriminant-specification	formal-discriminant-part
discriminant-specifier	schema-discriminant
discriminant-value	actual-discriminant-part
discriminated-schema	ordinal-type
	type-denoter
domain-type	new-pointer-type

– e –

else-part	if-statement
empty-statement	simple-statement
entire-function-access	function-access
entire-variable	control-variable
	variable-access
enumerated-type	new-ordinal-type
exponentiating-operator	factor
export-clause	export-list
export-list	export-part
export-part	interface-specification-part
export-range	export-list
export-renaming-clause	export-clause
exportable-name	export-clause
	export-renaming-clause
expression	actual-parameter
	assignment-statement
	Boolean-expression

	field-designator-identifier
	field-identifier
	function-heading
	function-identifier
	identifier-list
	import-renaming-clause
	imported-interface-identifier
	index-type-specification
	interface-identifier
	module-heading
	module-identifier
	parameter-identifier
	procedure-heading
	procedure-identifier
	program-heading
	result-variable-specification
	schema-definition
	schema-discriminant-identifier
	schema-identifier
	tag-field
	type-definition
	type-identifier
	variable-identifier
identifier-list	discriminant-specification
	enumerated-type
	module-parameter-list
	program-parameter-list
	record-section
	value-conformant-array-specification
	value-parameter-specification
	variable-conformant-array-specification
	variable-declaration
	variable-parameter-specification
if-statement	conditional-statement
implementation-directive	module-identification
import-clause	import-list
import-list	import-qualifier
import-part	block
	module-block
	module-heading
import-qualifier	import-specification
import-renaming-clause	import-clause
import-specification	import-part
imported-interface-identifier	constant-name
	function-name
	procedure-name

procedure-heading	procedural-parameter-specification
	procedure-and-function-heading-part
	procedure-declaration
procedure-identification	procedure-declaration
procedure-identifier	procedure-identification
	procedure-name
procedure-name	actual-parameter
	exportable-name
	procedure-statement
procedure-statement	simple-statement
program-block	program
program-component	program-block
program-heading	main-program-declaration
program-parameter-list	program-heading

– r –

read-parameter-list	procedure-statement
readln-parameter-list	procedure-statement
readstr-parameter-list	procedure-statement
real-type-name	simple-type
record-constant	field-designated-constant
record-function	record-function-access
record-function-access	component-function-access
record-section	fixed-part
record-type	unpacked-structured-type
record-type-name	structured-type-name
	structured-value-constructor
record-value	component-value
	structured-value-constructor
record-variable	field-designator
relational-operator	expression
remote-directive	function-declaration
	procedure-declaration
repeat-statement	repetitive-statement
repetitive-statement	structured-statement
restricted-type	new-type
result-type	function-heading
result-variable-specification	function-heading

– s –

scale-factor	number

```
                                      unsigned-real
schema-definition ................... type-definition-part
schema-discriminant ................. primary
schema-discriminant-identifier ...... schema-discriminant
schema-identifier ................... schema-name
schema-name ......................... discriminated-schema
                                      domain-type
                                      exportable-name
                                      parameter-form
                                      schema-definition
selective-import-option ............. import-qualifier
sequence-iteration .................. iteration-clause
set-constructor ..................... primary
                                      set-value
set-expression ...................... set-member-iteration
set-member-iteration ................ iteration-clause
set-type ............................ unpacked-structured-type
set-type-name ....................... structured-type-name
                                      structured-value-constructor
set-value ........................... structured-value-constructor
sign ................................ number
                                      scale-factor
                                      signed-integer
                                      signed-real
                                      simple-expression
signed-integer ...................... signed-number
signed-number ....................... number
signed-real ......................... signed-number
simple-expression ................... expression
simple-statement .................... statement
statement ........................... case-list-element
                                      else-part
                                      finalization-part
                                      for-statement
                                      if-statement
                                      initialization-part
                                      statement-sequence
                                      while-statement
                                      with-statement
statement-part ...................... block
statement-sequence .................. case-statement-completer
                                      compound-statement
                                      repeat-statement
string-character .................... string-element
string-constant ..................... indexed-constant
                                      substring-constant
```

– **t** –

– u –

– v –

```
variable-declaration ................ variable-declaration-part
variable-declaration-part ........... block
                                      module-block
                                      module-heading
variable-identifier ................. variable-name
variable-name ....................... entire-variable
                                      exportable-name
                                      type-inquiry-object
variable-parameter-specification ..... formal-parameter-section
variant-denoter ..................... variant-list-element
                                      variant-part-completer
variant-list-element ................ variant-part
variant-part ........................ field-list
variant-part-completer .............. variant-part
variant-part-value .................. field-list-value
variant-selector .................... variant-part
```

– w –

```
while-statement ..................... repetitive-statement
with-element ........................ with-list
with-list ........................... with-statement
with-statement ...................... structured-statement
word-symbol ......................... special-symbol
write-parameter ..................... write-parameter-list
                                      writeln-parameter-list
                                      writestr-parameter-list
write-parameter-list ................ procedure-statement
writeln-parameter-list .............. procedure-statement
writestr-parameter-list ............. procedure-statement
```

Annex B

(Informative)

Incompatibilities with Pascal standards

Programs that conform to the existing Pascal standards ISO 7185, BS 6192, and ANSI/IEEE770X3.97-1983 may need to have some identifiers changed in them because of the addition of new word-symbols in Extended Pascal. The new word-symbols that have been added to Extended Pascal are:

and_then	only	protected
bindable	or_else	qualified
export	otherwise	restricted
import	pow	value
module		

Annex C

(Informative)

Required identifiers

Identifier	Definition
abs	6.7.6.2
arctan	6.7.6.2
arg	6.7.6.2
bind	6.7.5.6
binding	6.7.6.8
BindingType	6.4.3.4
Boolean	6.4.2.2 c)
bound	6.4.3.4
capacity	6.4.3.3.3
card	6.7.6.3
char	6.4.2.2 d)
chr	6.7.6.4
cmplx	6.7.6.3
complex	6.4.2.2 e)
cos	6.7.6.2
date	6.7.6.9
DateValid	6.4.3.4
day	6.4.3.4
dispose	6.7.5.3
empty	6.7.6.5
eof	6.7.6.5
eoln	6.7.6.5
epsreal	6.4.2.2 b)
EQ	6.7.6.7
exp	6.7.6.2
extend	6.7.5.2
false	6.4.2.2 c)
GE	6.7.6.7
get	6.7.5.2
GetTimeStamp	6.7.5.8
GT	6.7.6.7
halt	6.7.5.7
hour	6.4.3.4
im	6.7.6.2
index	6.7.6.7
input	6.10, 6.11.4.2
integer	6.4.2.2 a)
LastPosition	6.7.6.6
LE	6.7.6.7
month	6.4.3.4
name	6.4.3.4
NE	6.7.6.7
new	6.7.5.3
odd	6.7.6.5
ord	6.7.6.4
output	6.10, 6.11.4.2
pack	6.7.5.4
page	6.10.5
polar	6.7.6.3
position	6.7.6.6
pred	6.7.6.4
put	6.7.5.2
re	6.7.6.2
read	6.7.5.2, 6.10.1
readln	6.10.2
readstr	6.7.5.5
real	6.4.2.2 b)
reset	6.7.5.2
rewrite	6.7.5.2
round	6.7.6.3
second	6.4.3.4
SeekRead	6.7.5.2
SeekUpdate	6.7.5.2
SeekWrite	6.7.5.2
sin	6.7.6.2
sqr	6.7.6.2
sqrt	6.7.6.2
StandardInput	6.11.4.2
StandardOutput	6.11.4.2
string	6.4.3.3.3
substr	6.7.6.7
succ	6.7.6.4
text	6.4.3.6
time	6.7.6.9
TimeStamp	6.4.3.4
TimeValid	6.4.3.4
trim	6.7.6.7
true	6.4.2.2 c)

Annex D

(Informative)

Errors and dynamic-violations

A complying processor is required to provide documentation concerning its treatment of errors. To facilitate the production of such documentation, all the errors specified in clause **6** are described again in this annex. Here, when a value of type T2 is said to be assignment-compatibility-erroneous with respect to a type T1, it means that one of the following four statements is true (see **6.4.6**).

a) 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) T1 and T2 are compatible set-types, and a member of the value of type T2 is not in the closed interval specified by the base-type of the type T1.

c) T1 and T2 are compatible, T1 is a string-type or a char-type, and the length of the value of T2 is greater than the capacity of T1.

d) T1 and T2 are produced from the same schema, but not with the same tuple.

Statements a) through c) describe errors; statement d) describes a dynamic-violation.

D.1 6.4.2.2
It is an error if an integer operation or function is not performed according to the mathematical rules for integer arithmetic.

D.2 6.4.2.4
It is a dynamic-violation if the smallest value of a subrange-type is greater than the largest value of the subrange-type when either subrange-bound is not nonvarying or contains a discriminant-identifier.

D.3 6.4.3.4
It is a dynamic-violation to attribute a value to the selector other than the value corresponding to the discriminant-identifier according to the tuple.

D.4 6.4.3.6
If f is a direct-access file-type with index-type T, a is the smallest value of type T, and b is the largest value of type T, it is an error if f.L and f.R are defined and length(f.L~f.R) > ord(b)-ord(a)+1.

D.5 6.4.7
It is an error if, within the activation, the domain of a schema contained by the activation is empty.

D.6 6.4.8
It is a dynamic-violation if the tuple consisting of the values denoted by the discriminant-values of the actual-discriminant-part taken in textual order is not in the domain of the schema.

D.7 6.5.1

The execution of any action, operation, or function, defined to operate on a variable, is an error if the variable is bindable and, as a result of the binding, the execution cannot be completed as defined.

D.8 6.5.3.2
It is an error if for an indexed-variable closest-containing an array-variable and a single index-expression, the value of the index-expression is assignment-compatibility-erroneous with respect to the index-type of the array-type.

D.9 6.5.3.2
It is an error if the value of the index-expression in an indexed-variable closest-containing a string-variable is less than one or greater than the length of the value of the string-variable.

D.10 6.5.3.2
It is an error to alter the length of the value of a string-variable when a reference to a component of the string-variable exists.

D.11 6.5.3.2
It is an error to access an indexed-variable when the string-variable, if any, of the indexed-variable is undefined.

D.12 6.5.3.3
It is an error unless a variant is active for the entirety of each reference and access to each component of the variant of a record-variable.

D.13 6.5.4
It is an error if the pointer-variable of an identified-variable either denotes a nil-value or is undefined.

D.14 6.5.4
It is an error to remove from its pointer-type the identifying-value of an identified-variable when a reference to the identified-variable exists.

D.15 6.5.5
It is an error to alter the value of a file-variable f when a reference to the buffer-variable f↑exists.

D.16 6.5.6
It is an error if the string-variable of the substring-variable is undefined, or if the value of an index-expression in a substring-variable is less than 1 or greater than the length of the value of the string-variable of the substring-variable, or if the value of the first index-expression is greater than the value of the second index-expression.

D.17 6.5.6
It is an error to alter the length of the value of a string-variable when a reference to a substring of the string-variable exists.

D.18 6.7.3.2
It is an error if the value of any actual value parameter is assignment-compatibility-erroneous with respect to the type possessed by the corresponding formal-parameter.

D.19 6.7.3.2
It is an error if the actual-parameters contained by the activation-point of an activation corresponding to formal-parameters that occur in a single value-parameter-specification containing a schema-name that denotes the schema denoted by the required schema-identifier string are string-type or char-type values with differing lengths.

D.20 6.7.3.2
It is a dynamic-violation if the underlying-types of the types of the actual-parameters corresponding to a parameter-form, that is in a value-parameter-specification and that contains a schema-name not denoting the schema string, are not produced from a schema with the same tuple.

D.21 6.7.3.3
It is a dynamic-violation if the underlying-types of the types possessed by the formal-parameter and actual-parameter are produced from the same schema, but not with the same tuple.

D.22 6.7.3.5
It is a dynamic-violation if the type denoted by the result-type closest-contained by the formal-parameter-section and the type of the function are produced from the same schema, but not with the same tuple.

D.23 6.7.3.6
It is a dynamic-violation if the type-name in the parameter-form of each value-parameter-specification denotes a type produced from the same schema but not with the same tuple.

D.24 6.7.3.6
It is a dynamic-violation if the type-name in the parameter-form of each variable-parameter-specification denotes a type produced from the same schema but not with the same tuple.

D.25 6.7.3.7.1
It is an error if the conformant-actual-variables corresponding to formal-parameters that occur in a single value-conformant-array-specification possess fixed-string-types that have different capacities or that are not conformable with the conformant-array-form.

D.26 6.7.3.7.2
It is an error if the value parameter is assignment-compatibility-erroneous with respect to the type possessed by the conformant-actual-variable.

D.27 6.7.3.8
At any place where the rule of conformability is used, it is an error if the smallest or largest value specified by the index-type of an array-type with a single index-type lies outside the closed interval specified by the type denoted by the ordinal-type-name of the index-type-specification of a conformant-array-form closest-containing a single index-type-specification.

D.28 6.7.5.2
When extend(f) is activated, it is an error if either f0.L or f0.R is undefined.

D.29 6.7.5.2
When put(f) is activated, it is an error if f0.M is not Generation or Update; if either f0.L or f0.R is undefined; if f0↑ is undefined; or if f0.R <> S( ), and f is not a direct-access file-type.

D.30 6.7.5.2
When update(f) is activated, it is an error if f0.M is not Generation or Update, if either f0.L or f0.R is undefined, if f0↑ is undefined, or if f is not a direct-access file-type.

D.31 6.7.5.2
When reset(f) is activated, it is an error if either f0.L or f0.R is undefined.

D.32 6.7.5.2
When get(f) is activated, it is an error if f0.M is not Inspection or Update, if either f0.L or f0.R is undefined, or if f0.R = S( ).

D.33 6.7.5.2
When SeekWrite(f,n) is activated, it is an error if either f0.L or f0.R is undefined, if (ord(n)-ord(a)) < 0, if length(f0.L~f0.R) < (ord(n)-ord(a)), or if n is assignment-compatibility-erroneous with respect to T.

D.34 6.7.5.2
When SeekRead(f,n) is activated, it is an error if either f0.L or f0.R is undefined, if (ord(n)-ord(a)) < 0, if length(f0.L~f0.R) < (ord(n)-ord(a)), or if n is assignment-compatibility-erroneous with respect to T.

D.35 6.7.5.2
When SeekUpdate(f,n) is activated, it is an error if either f0.L or f0.R is undefined, if (ord(n)-ord(a)) < 0, if length(f0.L~f0.R) < (ord(n)-ord(a)), or if n is assignment-compatibility-erroneous with respect to T.

D.36 6.7.5.2
When read(f,v) is activated, with f, a file-variable possessing a type other than that denoted by text, and v, a variable-access, it is an error if f0.M is not Inspection or Update, if either f0.L or f0.R is undefined, if f0.R = S( ), or if the value of f↑ is assignment-compatibility-erroneous with respect to the type of v.

D.37 6.7.5.2
When write(f,e) is activated, with f, a file-variable possessing a type other than that denoted by text, and e, an expression, it is an error if f0.M is not either Generation or Update; if either f0.L or f0.R is undefined; if e is undefined; if f0.R <> S( ), and f is not a direct-access file; or if the value of the expression e is assignment-compatibility-erroneous with respect to the component-type of the file-type of f.

D.38 6.7.5.3
When new(p,c_1,...,c_n) is activated, it is an error if a variant of a variant-part within the new variable becomes active, and a different variant of the variant-part is one of the specified variants.

D.39 6.7.5.3
When new(p,d_1,...,d_s) is activated, it is a dynamic-violation if the tuple consisting of the values of the expressions d_1,...,d_s taken in textual order is not in the domain of the schema denoted by the schema-identifier of the domain-type of the pointer-type possessed by p.

D.40 6.7.5.3

When dispose(q) is activated, it is an error if the identifying-value had been created using the form new(p,c_1,...,c_n).

D.41 6.7.5.3
When dispose(q,k_1,...,k_m) is activated, it is an error unless the variable had been created using the form new(p,c_1,...,c_n) and m is equal to n.

D.42 6.7.5.3
When dispose(q,k_1,...,k_m) is activated, it is 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.

D.43 6.7.5.3
When either dispose(q) or dispose(q,k_1,...,k_m) is activated, it is an error if q has a nil-value or is undefined.

D.44 6.7.5.3
It is an error if a variable created using the form new(p,c_1,...,c_n) is accessed by the identified-variable of the variable-access of a primary, of an assignment-statement, or of an actual-parameter.

D.45 6.7.5.4
When pack(a,i,z) is activated, it is an error if the value of i is assignment-compatibility-erroneous with respect to the index-type of the type of a.

D.46 6.7.5.4
When pack(a,i,z) is activated, it is an error if any of the components of a are both undefined and accessed.

D.47 6.7.5.4
When pack(a,i,z) is activated, it is an error if the index-type of the type of a is exceeded.

D.48 6.7.5.4
When unpack(z,a,i) is activated, it is an error if the value of i is assignment-compatibility-erroneous with respect to the index-type of the type of a.

D.49 6.7.5.4
When unpack(z,a,i) is activated, it is an error if any of the components of z are undefined.

D.50 6.7.5.4
When unpack(z,a,i) is activated, it is an error if the index-type of the type of a is exceeded.

D.51 6.7.5.5
When readstr(e,v_1,...,v_n) is activated, it is an error if the equivalent of eof(f) is true upon completion.

D.52 6.7.5.5
When writestr(s,p_1,...,p_n) is activated, it is an error if any of the write-parameters accesses the referenced string-variable.

D.53 6.7.5.5

When writestr(s,p_1,...,p_n) is activated, it is an error if the equivalent of eoln(f) is false upon completion.

D.54 6.7.5.6
When bind(f,b) is activated, it is a dynamic-violation if the variable f is already bound to an external entity.

D.55 6.7.5.6
When bind(f,b) is activated, it is a dynamic-violation if the variable f, which possesses a file-type, does not possess the bindability that is bindable.

D.56 6.7.5.6
When unbind(f) is activated, it is a dynamic-violation if the variable f, which possesses a file-type, does not possess the bindability that is bindable.

D.57 6.7.6.2
From the value of integer-type or real-type of the expression x, sqr(x) computes a value, if any, of the same type, for the square of x. When sqr(x) is activated, it is an error if no such value exists.

D.58 6.7.6.2
From the value of real-type of the expression x, ln(x) computes a value of real-type for the natural logarithm of x, if x is greater than zero. When ln(x) is activated, it is an error if the value of the expression x is not greater than zero.

D.59 6.7.6.2
From the value of complex-type of the expression x, ln(x) computes a value of complex-type for the natural logarithm of x, if x is not equal to zero. When ln(x) is activated, it is an error if the value of the expression x is equal to zero.

D.60 6.7.6.2
From the value of real-type of the expression x, sqrt(x) computes a value of real-type for the non-negative square root of x, if x is not negative. When sqrt(x) is activated, it is an error if the value of the expression x is negative.

D.61 6.7.6.3
From the value of real-type of the expression x, trunc(x) computes a value, if any, of integer-type, such that if x is positive or zero then $0 <= x - \text{trunc}(x) < 1$; otherwise, $-1 < x - \text{trunc}(x) <= 0$. When trunc(x) is activated, it is an error if no such value exists.

D.62 6.7.6.3
From the value of real-type of the expression x, round(x) computes a value, if any, of integer-type, such that if x is positive or zero then round(x) is equivalent to trunc(x+0.5); otherwise, round(x) is equivalent to trunc(x−0.5). When round(x) is activated, it is an error if no such value exists.

D.63 6.7.6.3
From the value of unpacked-canonical-set-of-T-type or packed-canonical-set-of-T-type of the expression x, card(x) computes a value of integer-type for the number of members in x. When card(x) is activated, it is an error if no such value exists.

D.64 6.7.6.4
From the value of integer-type of the expression x, chr(x) computes a value, if any, of char-type that is the value whose ordinal number is equal to the value of the expression x. When chr(x) is activated, it is an error if no such value exists.

D.65 6.7.6.4
From the value of ordinal-type of the expression x, and the value of integer-type of the expression k, succ(x,k) computes a value, if any, of the same ordinal-type whose ordinal number is ord(x) + k. When succ(x,k) is activated, it is an error if no such value exists.

D.66 6.7.6.5
When eof(f) is activated, it is an error if f is undefined.

D.67 6.7.6.5
When eoln(f) is activated, it is an error if f is undefined or if eof(f) is true.

D.68 6.7.6.5
When empty(f) is activated, it is an error if f is undefined.

D.69 6.7.6.6
When position(f) is activated, it is an error if f is undefined or if the value of position(f) does not exist.

D.70 6.7.6.6
When LastPosition(f) is activated, it is an error if f is undefined or if the value of LastPosition(f) does not exist.

D.71 6.7.6.7
When substr(s,i,j) is activated, it is an error if the value of i is less than or equal to zero, if the value of j is less than zero, or if the value of (i)+(j)−1 is greater than the length of the value of s.

D.72 6.7.6.8
When binding(f) is activated, it is a dynamic-violation if the variable f, which possesses a file-type, does not possess the bindability that is bindable.

D.73 6.7.6.9
When date(t) is activated, it is an error if the values of the fields **Day**, **Month**, and **Year** of the value of t do not represent a valid calendar date.

D.74 6.8.1
It is an error to compute the value of an expression that contains a variable denoted by a variable-access of a primary that is undefined at the time of its use.

D.75 6.8.3.2
For a term of the form x/y, it is an error if the value of y is zero.

D.76 6.8.3.2
For a term of the form i div j, it is an error if the value of j is zero.

D.77 6.8.3.2

For a term of the form i mod j, it is an error if the value of j is zero or negative.

D.78 6.8.3.2
For a factor of the form x**y, it is an error if x is zero and y is less than or equal to zero.

D.79 6.8.3.2
For a factor of the form x**y, where x is of integer-type or real-type, it is an error if x is negative.

D.80 6.8.3.2
For a factor of the form x pow y, it is an error if x is zero and y is less than or equal to zero.

D.81 6.8.5
It is an error if the value of the result of an activation of a function is undefined upon completion of the algorithm of the activation.

D.82 6.8.6.2
It is an error if for an indexed-function-access closest-containing an array-function and a single index-expression, the value of the index-expression is assignment-compatibility-erroneous with respect to the index-type of the array-type possessed by the result of the array-function.

D.83 6.8.6.2
It is an error if the value of the index-expression in an indexed-function-access closest-containing a string-function is less than one or greater than the length of the value of the string-function.

D.84 6.8.6.3
It is an error to denote a component of an inactive variant of a variant-part of a record-function.

D.85 6.8.6.4
It is an error if the pointer-function of a function-identified-variable denotes the nil-value.

D.86 6.8.6.5
It is an error if the value of an index-expression in a substring-function-access is less than one or greater than the length of the value of the string-function of the substring-function-access or if the value of the first index-expression is greater than the value of the second index-expression.

D.87 6.8.7.1
It is an error if the value denoted by an expression in a component-value is assignment-compatibility-erroneous with respect to the type of the component-value of a structured-value-constructor.

D.88 6.8.8.2
It is an error if for an indexed-constant closest-containing an array-constant and a single index-expression, the value of the index-expression is assignment-compatibility-erroneous with respect to the index-type of the array-type.

D.89 6.8.8.2
It is an error if the value of the index-expression in an indexed-constant closest-containing a string-constant is less than one or greater than the length of the value of the string-constant.

D.90 6.8.8.3

It is an error to access a component of an inactive variant of a variant-part of a record-constant.

D.91 6.8.8.4
It is an error if the value of an index-expression in a substring-constant is less than 1 or greater than the length of the value of the string-constant of the substring-constant or if the value of the first index-expression is greater than the value of the second index-expression.

D.92 6.9.2.2
It is an error if the value of the expression of an assignment-statement is assignment-compatibility-erroneous with respect to the type possessed by either the variable denoted by the variable-access of the assignment statement, or by the activation result that is denoted by the function-identifier of the assignment-statement.

D.93 6.9.2.4
It is a dynamic-violation if the commencement of the activation containing the program-point has not completed.

D.94 6.9.3.5
For a case-statement without a case-statement-completer, it is a dynamic-violation if no case-range closest-contained by a case-list-element of the case-statement denotes the value of the case-index upon execution of the case-statement.

D.95 6.9.3.9.2
It is an error if the value of initial-value or the value of final-value of a sequence-iteration of an iteration-clause of a for-statement is assignment-compatibility-erroneous with respect to the type possessed by the control-variable of the for-statement, if the statement of the for-statement is executed.

D.96 6.9.3.9.3
It is an error if any value that is a member of the value of the set-expression of a set-member-iteration of an iteration-clause of a for-statement is assignment-compatibility-erroneous with respect to the type possessed by the control-variable of the for-statement.

D.97 6.10.1
When read is applied to textfile f, it is an error if the buffer variable f↑ is undefined, if f0.M is not Inspection or Update, if either f0.L or f0.R is undefined, or if f0.R=S( ).

D.98 6.10.1
On reading an integer from a textfile, after skipping preceding spaces and end-of-lines, it is an error if the rest of the sequence of characters does not form a signed-integer.

D.99 6.10.1
On reading an integer from a textfile, it is an error if the value of the signed-integer read is assignment-compatibility-erroneous with respect to the type possessed by the variable-access.

D.100 6.10.1
On reading a number from a textfile, after skipping preceding spaces and end-of-lines, it is an error if the rest of the sequence of characters does not form a number.

D.101 6.10.3

When write is applied to textfile f, it is an error if f is undefined or if f.M is Inspection.

D.102 6.10.3.1
For write-parameters of the form e:TotalWidth or of the form e:TotalWidth:FracDigits, it is an error if the value of TotalWidth is less than zero.

D.103 6.10.3.1
For write-parameters of the form e:TotalWidth:FracDigits, it is an error if the value of FracDigits is less than zero.

D.104 6.10.4
When writeln(f) is activated, it is an error if f0 is undefined, if f0.M is not Generation, or if f0.R <> S( ).

D.105 6.10.5
When page(f) is activated, it is an error if f0 is undefined, if f0.M is not Generation, or if f0.R <> S( ).

Annex E

(Informative)

Implementation-defined features

A complying processor is required to provide a definition of all the implementation-defined features of the language (see **5.1 d)**). To facilitate the production of this definition, all the implementation-defined features specified in clause **6** are listed again in this annex.

E.1 6.1.9
The 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 is implementation-defined.

E.2 6.1.11
Provision of the reference tokens ↑, [, and], of the alternative token @, and of the delimiting characters { and }, is implementation-defined.

E.3 6.4.2.2 a)
The value of integer-type denoted by the required constant-identifier **maxint** is implementation-defined.

E.4 6.4.2.2 b)
The values of real-type are implementation-defined approximations to an implementation-defined subset of the real numbers.

E.5 6.4.2.2 b)
The value of real-type denoted by the required constant-identifier **minreal** is implementation-defined.

E.6 6.4.2.2 b)
The value of real-type denoted by the required constant-identifier **maxreal** is implementation-defined.

E.7 6.4.2.2 b)
The value of real-type denoted by the required constant-identifier **epsreal** is implementation-defined.

E.8 6.4.2.2 b)
The results of the real arithmetic operators and functions are approximations to the corresponding mathematical results. The accuracy of this approximation is implementation-defined.

E.9 6.4.2.2 d)
The values of char-type are the enumeration of a set of characters that is implementation-defined.

E.10 6.4.2.2 d)
The ordinal numbers of the character values are values of integer-type that are implementation-defined.

E.11 6.4.2.2 d)
The value of char-type denoted by the required constant-identifier **maxchar** is implementation-defined.

E.12 6.4.2.2 e)
The values of complex-type are implementation-defined approximations to an implementation-defined subset of the complex numbers.

E.13 6.4.2.2 e)
The results of the complex arithmetic operators and required functions are approximations to the corresponding mathematical results. The accuracy of this approximation is implementation-defined.

E.14 6.4.3.4
The variable-string-type of the field associated with the required field-identifier **name** in the required record-type denoted by the required type-identifier **BindingType** is implementation-defined.

E.15 6.7.5.2
The activities on external entities to which file-variables are bound and the point at which they are actually performed are implementation-defined.

E.16 6.7.5.6
The binding as a result of the statement bind(f,b) of a variable denoted by f to an entity that is external to the program and that is designated by b is implementation-defined.

E.17 6.7.5.8
The meaning of "current date" returned by the procedure **GetTimeStamp** is implementation-defined.

E.18 6.7.5.8
The meaning of "current time" returned by the procedure **GetTimeStamp** is implementation-defined.

E.19 6.7.6.8
The value returned by binding(f), where f is a variable, is implementation-defined.

E.20 6.7.6.9
The length of the string returned by the function **date** is implementation-defined.

E.21 6.7.6.9
The representation that is returned by date(t) of the calendar date denoted by t is implementation-defined.

E.22 6.7.6.9
The length of the string returned by the function **time** is implementation-defined.

E.23 6.7.6.9
The representation that is returned by time(t) of the time denoted by t is implementation-defined.

E.24 6.10.3.1
The default value of TotalWidth for integer-type is implementation-defined.

E.25 6.10.3.1
The default value of TotalWidth for real-type is implementation-defined.

E.26 6.10.3.1
The default value of TotalWidth for Boolean-type is implementation-defined.

E.27 6.10.3.4.1
The value of ExpDigits is implementation-defined.

E.28 6.10.3.4.1
The value of the exponent character ('e' or 'E') is implementation-defined.

E.29 6.10.3.5
The case of each letter of `'True'` and `'False'` for output of Boolean values is implementation-defined.

E.30 6.10.5
The effect of the statement page(f) on the textfile f is implementation-defined.

E.31 6.11.1
The binding of variables denoted by the module parameters to entities external to the program is implementation-defined.

E.32 6.11.4.2
The effect of the application of either of the required procedures **reset**, **rewrite**, or **extend** to the required textfile **input** is implementation-defined.

E.33 6.11.4.2
The effect of the application of either of the required procedures **reset**, **rewrite**, or **extend** to the required textfile **output** is implementation-defined.

E.34 6.12
The binding of variables denoted by the program parameters to entities external to the program is implementation-defined.

Annex F

(Informative)

Implementation-dependent features

A complying processor is required to provide documentation concerning its treatment of all the implementation-dependent features of the language (see **5.1 j)** and **5.1 g)**). To facilitate the production of this definition, all the implementation-dependent features specified in clause **6** are listed again in this annex.

F.1 6.4.8
A discriminant-value that is an expression shall denote the current value of the expression upon activation of the block closest-containing the discriminant-value. The order of evaluation of all such expressions contained in that block is implementation-dependent.

F.2 6.5.3.2
The order of both the evaluation of the index-expressions of, and the access to the array-variable or string-variable of, an indexed-variable is implementation-dependent.

F.3 6.5.6
The order of both the evaluation of the index-expressions of, and the access to the string-variable of, a substring-variable is implementation-dependent.

F.4 6.7.5.3
The order of evaluation of the expressions d_1, ...,d_s in new(p,d_1, ...,d_s) is implementation-dependent.

F.5 6.8.1
The order of evaluation of the expressions of a member-designator is implementation-dependent.

F.6 6.8.1
The order of evaluation of the member-designators of a set-constructor is implementation-dependent.

F.7 6.8.3.1
Except for and_then and or_else, the order of evaluation of the operands of a dyadic operator is implementation-dependent.

F.8 6.8.5
The order of evaluation, accessing, and binding of the actual-parameters of the actual-parameter-list, if any, of a function-designator is implementation-dependent.

F.9 6.8.6.2
The order of evaluation both of the index-expressions of, and of the array-function or string-function of, an indexed-function-access is implementation-dependent.

F.10 6.8.6.5
The order of evaluation both of the index-expressions of, and of the string-function of, a substring-function-access is implementation-dependent.

F.11 6.8.7.1
The order of evaluation of the component-values contained by a structured-value-constructor is implementation-dependent.

F.12 6.8.8.2
The order of evaluation of the index-expressions of an indexed-constant is implementation-dependent.

F.13 6.8.8.4
The order of evaluation of the index-expressions of a substring-constant is implementation-dependent.

F.14 6.9.2.2
The order of accessing the variable and evaluating the expression of an assignment-statement is implementation-dependent.

F.15 6.9.2.3
The order of evaluation, accessing, and binding of the actual-parameters of the actual-parameter-list, if any, of a procedure-statement is implementation-dependent.

F.16 6.9.3.9.3
In a set-member-iteration, the order of selection of members of the value of a set-expression is implementation-dependent.

F.17 6.10.1
When reading an integer or real representation from a textfile f, it is implementation-dependent whether the first character examined is the value of the buffer-variable or the value of the first component of f.R.

F.18 6.10.5
The effect of inspecting a textfile to which the **page** procedure was applied during generation is implementation-dependent.

Annex G

(Informative)

Bibliography

ANSI X3.30-1985, Representation of Calendar Date and Ordinal Date for Information Interchange.

ANSI/IEEE770X3.97-1983, American National Standard Pascal Computer Programming Language.

BS6192:1982, British Standard Specification for Computer programming language Pascal.

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

ISO 6903:1985, Information processing — Representation of numerical values in character strings for information interchange.

ISO 7185:1983, Programming languages — PASCAL.

ISO 8601:1988, Data elements and interchange formats — Information interchange — Representation of dates and times.

Index