

INTERNATIONAL STANDARD

ISO/IEC
10206

First edition
1991-04-15

Information technology — Programming languages — Extended Pascal

Langages de programmation — Pascal étendu



Reference number
ISO/IEC 10206:1991(E)
Documents
Document Sharing Hub

Contents	Page
Introduction	vi
1 Scope	1
2 Normative reference	1
3 Definitions	1
3.1 Dynamic-violation	2
3.2 Error	2
3.3 Extension	2
3.4 Implementation-defined	2
3.5 Implementation-dependent	2
3.6 Processor	2
4 Definitional conventions	2
5 Compliance	3
5.1 Processors	4
5.2 Programs	5
6 Requirements	5
6.1 Lexical tokens	5
6.1.1 General	5
6.1.2 Special-symbols	6
6.1.3 Identifiers	6
6.1.4 Remote-directives	6
6.1.5 Interface-directives	7
6.1.6 Implementation-directives	7
6.1.7 Numbers	7
6.1.8 Labels	8

6.1.9	Character-strings	8
6.1.10	Token separators	9
6.1.11	Lexical alternatives	9
6.2	Blocks, scopes, activations, and states	10
6.2.1	Blocks	10
6.2.2	Scopes	10
6.2.3	Activations	13
6.2.4	States	15
6.3	Constants	16
6.3.1	General	16
6.3.2	Example of a constant-definition-part	17
6.4	Types and schemata	17
6.4.1	Type-definitions	17
6.4.2	Simple-types	19
6.4.3	Structured-types	24
6.4.4	Pointer-types	32
6.4.5	Compatible types	33
6.4.6	Assignment-compatibility	33
6.4.7	Schema-definitions	34
6.4.8	Discriminated-schemata	35
6.4.9	Type-inquiry	36
6.4.10	Example of a type-definition-part	36
6.5	Declarations and denotations of variables	38
6.5.1	Variable-declarations	38
6.5.2	Entire-variables	39
6.5.3	Component-variables	39
6.5.4	Identified-variables	41
6.5.5	Buffer-variables	41
6.5.6	Substring-variables	42
6.6	Initial states	42
6.7	Procedure and function declarations	43
6.7.1	Procedure-declarations	43
6.7.2	Function-declarations	45
6.7.3	Parameters	48
6.7.4	Required procedures and functions	56
6.7.5	Required procedures	56
6.7.6	Required functions	63
6.8	Expressions	69
6.8.1	General	69
6.8.2	Constant-expressions	71

6.8.3	Operators	71
6.8.4	Schema-discriminants	75
6.8.5	Function-designators	76
6.8.6	Function-accesses	76
6.8.7	Structured-value-constructors	78
6.8.8	Constant-accesses	80
6.9	Statements	82
6.9.1	General	82
6.9.2	Simple-statements	82
6.9.3	Structured-statements	84
6.9.4	Threats	90
6.10	Input and output	91
6.10.1	The procedure read	91
6.10.2	The procedure readln	92
6.10.3	The procedure write	93
6.10.4	The procedure writeln	96
6.10.5	The procedure page	97
6.11	Modules	97
6.11.1	Module-declarations	97
6.11.2	Export-part	99
6.11.3	Import-specifications	100
6.11.4	Required interfaces	102
6.11.5	Example of a module	102
6.11.6	Examples of program-components that are module-declarations	104
6.11.7	Example of exporting a range of enumerated-type values	111
6.12	Main-program-declarations	112
6.13	Programs	114
Annexes		
Annex A	Collected syntax	116
Annex B	Incompatibilities with Pascal standards	176
Annex C	Required identifiers	177
Annex D	Errors and dynamic-violations	178
Annex E	Implementation-defined features	186
Annex F	Implementation-dependent features	189
Annex G	Bibliography	191
Index		192

Foreword

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

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

International Standard ISO/IEC 10206 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*.

Annexes A to G are for information only.

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.

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.

Table 1 — Metalinguage symbols

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

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 $y_1, y_2, \dots$, 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 `il.x` (see 6.5.1) constructed from an interface-identifier `il` 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 il qualified only (x);
var y : integer;
begin
    il.x := ...
```

the construct `il.x` is allowed but `il.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 $k = (\text{ord}(n) - \text{ord}(m) + 1)$ 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

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

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

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

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 $(F_1 * F_2 * \dots * F_m)$.

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

$$(k, X_k)$$

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 $x \sim y$ 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 $S(c) \sim x$ and $x \sim S(c)$ shall be sequences, distinct from $S()$, of type S .
- c) Let c , S , and x be as in b), let y denote the sequence $S(c) \sim x$ and let z denote the sequence $x \sim S(c)$; 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 $x \sim (y \sim z) = (x \sim y) \sim z$, $S() \sim x = x$, and $x \sim S() = x$ 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 $f.L \sim f.R$ 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 $\text{length}(f.L \sim f.R) > \text{ord}(b) - \text{ord}(a) + 1$.

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 $I_1, I_2, \dots, I_n$, in order of occurrence of their defining-points, shall determine a set of *allowed* discriminant tuples of the form

$$(V_1, V_2, \dots, V_n)$$

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) = 1..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;
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 .