INTERNATIONAL STANDARD

ISO/IEC 7185

Second edition
1990-10-15

Information technology — Programming languages — Pascal

(Revision of ISO 7185 : 1983)

Technologies de l'information — Langages de programmation — Pascal

(Révision de l'ISO 7185 : 1983)

Reference number
ISO/IEC 7185 : 1990 (E)

AVA Documents
Document Sharing Hub

Full document details: ISO IEC 07185-1990

Contents

Page

1 Scope 1

2 Normative reference 1

3 Definitions 1

4 Definitional conventions 2

5 Compliance 3

5.1 Processors 3

5.2 Programs 5

6 Requirements 5

6.1 Lexical tokens 5

6.2 Blocks, scopes and activations 8

6.3 Constant - definitions 11

6.4 Type - definitions 11

6.5 Declarations and denotations of variables 21

6.6 Procedure and function declarations 24

6.7 Expressions 38

6.8 Statements 42

6.9 Input and output 49

6.10 Programs 55

Annexes

A Collected syntax 58

B Cross - references 64

C Required identifiers 71

D Errors 72

E Implementation - defined features 76

F Implementation - dependent features 78

© ISO/IEC 1990
All rights reserved. No part of this publication may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying and microfilm, without permission in writing from the publisher.
ISO/IEC Copyright Office • Case postale 56 • CH-1211 Genève 20 • Switzerland
Printed in Switzerland

Documents
Document Sharing Hub

Full document details: ISO IEC 07185-1990

Foreword

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

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

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

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

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

Documents
Document Sharing Hub

Full document details: ISO IEC 07185-1990

Introduction

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

Language history

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

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

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

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

In drafting this International Standard the continued stability of Pascal has been a prime objective. However, apart from changes to clarify the specification, two major changes have been introduced.

a) The syntax used to specify procedural and functional parameters has been changed to require the use of a procedure or function heading, as appropriate (see **6.6.3.1**); this change was introduced to overcome a language insecurity.

b) A fifth kind of parameter, the conformant-array-parameter, has been introduced (see **6.6.3.7**). With this kind of parameter, the required bounds of the index-type of an actual-parameter are not fixed, but are restricted to a specified range of values.

Project history

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

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

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

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

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

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

AVA Documents
Document Sharing Hub

Full document details: ISO IEC 07185-1990

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

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

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

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

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

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

d) in the handling of run-time errors.

Documents
Document Sharing Hub

Full document details: ISO IEC 07185-1990

Information technology — Programming languages — Pascal

1 Scope

1.1

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

1.2

This International Standard does not specify

a) the size or complexity of a program and its data that will exceed the capacity of any specific data processing system or the capacity of a particular processor, nor the actions to be taken when the corresponding limits are exceeded;

b) the minimal requirements of a data processing system that is capable of supporting an implementation of a processor for Pascal;

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

d) the mechanism by which programs written in Pascal are transformed for use by a data processing system;

e) the method for reporting errors or warnings;

f) the typographical representation of a program published for human reading.

2 Normative reference

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

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

3 Definitions

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

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

AVA Documents
Document Sharing Hub

Full document details: ISO IEC 07185-1990

3.1 Error

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

NOTES

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

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

3.2 Extension

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

3.3 Implementation-defined

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

3.4 Implementation-dependent

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

3.5 Processor

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

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

4 Definitional conventions

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

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

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

AVA Documents
Document Sharing Hub

Full document details: ISO IEC 07185-1990

Table 1 — Metalanguage symbols

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

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

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

—the x *of* a y: refers to the x occurring directly in a production defining y;

—the x *in* a y: is synonymous with 'the x of a y';

—a y *containing* an x: refers to any y from which an x is directly or indirectly derived;

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

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

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

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

Examples in this International Standard are equivalent to NOTES.

5 Compliance

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

5.1 Processors

A processor complying with the requirements of this International Standard shall

a) if it complies at level 0, accept all the features of the language specified in clause **6**, except for **6.6.3.6 e)**, **6.6.3.7**, and **6.6.3.8**, with the meanings defined in clause **6**;

b) if it complies at level 1, accept all the features of the language specified in clause **6** with the meanings defined in clause **6**;

Documents
Document Sharing Hub

Full document details: ISO IEC 07185-1990

c) not require the inclusion of substitute or additional language elements in a program in order to accomplish a feature of the language that is specified in clause **6**;

d) be accompanied by a document that provides a definition of all implementation-defined features;

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

f) treat each violation that is designated an error in at least one of the following ways

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

AVA Documents
Document Sharing Hub

Full document details: ISO IEC 07185-1990

5.2 Programs

A program conforming with the requirements of this International Standard shall

a) if it conforms at level 0, use only those features of the language specified in clause **6**, except for **6.6.3.6 e)**, **6.6.3.7**, and **6.6.3.8**;

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

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

NOTES

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

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

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

6 Requirements

6.1 Lexical tokens

NOTE — The syntax given in this subclause describes the formation of lexical tokens from characters and the separation of these tokens and therefore does not adhere to the same rules as the syntax in the rest of this International Standard.

6.1.1 General

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

```
letter = 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' | 'j'
       | 'k' | 'l' | 'm' | 'n' | 'o' | 'p' | 'q' | 'r' | 's' | 't'
       | 'u' | 'v' | 'w' | 'x' | 'y' | 'z' .

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

6.1.2 Special-symbols

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

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

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


AVA Documents
Document Sharing Hub

Full document details: ISO IEC 07185-1990

6.1.3 Identifiers

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

identifier = letter { letter | digit } .

Examples:

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

6.1.4 Directives

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

directive = letter { letter | digit } .

NOTE — Many processors provide, as an extension, the directive **external**, which is used to specify that the procedure-block or function-block corresponding to that procedure-heading or function-heading is external to the program-block. Usually it is in a library in a form to be input to, or that has been produced by, the processor.

6.1.5 Numbers

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

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

signed-real = [sign] unsigned-real .

signed-integer = [sign] unsigned-integer .

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

sign = '+' | '–' .

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

unsigned-integer = digit-sequence .

fractional-part = digit-sequence .

scale-factor = [sign] digit-sequence .

digit-sequence = digit { digit } .

AVA Documents
Document Sharing Hub

Full document details: ISO IEC 07185-1990

Examples:

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

6.1.6 Labels

Labels shall be digit-sequences and shall be distinguished by their apparent integral values and shall be in the closed interval 0 to 9999. The *spelling* of a label shall be its apparent integral value.

label = digit-sequence .

6.1.7 Character-strings

A character-string containing a single string-element shall denote a value of the required char-type (see **6.4.2.2**). A character-string containing more than one string-element shall denote a value of a string-type (see **6.4.3.2**) with the same number of components as the character-string contains string-elements. All character-strings with a given number of components shall possess the same string-type.

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

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

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

apostrophe-image = "'" .

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

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

Examples:

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

6.1.8 Token separators

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

('{' | '(*') commentary ('*)' | '}')

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

NOTES

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

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

AVA Documents
Document Sharing Hub

Full document details: ISO IEC 07185-1990

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

6.1.9 Lexical alternatives

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

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

The alternative representations for the tokens shall be

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

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

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

NOTE — 2 See also **1.2 f)**.

6.2 Blocks, scopes, and activations

6.2.1 Blocks

A block closest-containing a label-declaration-part in which a label occurs shall closest-contain exactly one statement in which that label occurs. The occurrence of a label in the label-declaration-part of a block shall be its defining-point for the region that is the block. Each applied occurrence of that label (see **6.2.2.8**) shall be a label. Within an activation of the block, all applied occurrences of that label shall denote the corresponding program-point in the algorithm of the activation at that statement (see **6.2.3.2 b)**).

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

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

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

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

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

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


Documents
Document Sharing Hub

Full document details: ISO IEC 07185-1990

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

statement-part = compound-statement .

6.2.2 Scopes

6.2.2.1

Each identifier or label contained by the program-block shall have a defining-point.

6.2.2.2

Each defining-point shall have a *region* that is a part of the program text, and a scope that is a part or all of that region.

6.2.2.3

The region of each defining-point is defined elsewhere (see **6.2.1**, **6.2.2.10**, **6.3**, **6.4.1**, **6.4.2.3**, **6.4.3.3**, **6.5.1**, **6.5.3.3**, **6.6.1**, **6.6.2**, **6.6.3.1**, **6.8.3.10**, **6.10**).

6.2.2.4

The scope of each defining-point shall be its region (including all regions enclosed by that region) subject to **6.2.2.5** and **6.2.2.6**.

6.2.2.5

When an identifier or label has a defining-point for region A and another identifier or label having the same spelling has a defining-point for some region B enclosed by A, then region B and all regions enclosed by B shall be excluded from the scope of the defining-point for region A.

6.2.2.6

The region that is the field-specifier of a field-designator shall be excluded from the enclosing scopes.

6.2.2.7

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

6.2.2.8

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

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

AVA Documents
Document Sharing Hub

Full document details: ISO IEC 07185-1990

6.2.2.9

The defining-point of an identifier or label shall precede all applied occurrences of that identifier or label contained by the program-block with one exception, namely that an identifier can have an applied occurrence in the type-identifier of the domain-type of any new-pointer-types contained by the type-definition-part containing the defining-point of the type-identifier.

6.2.2.10

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

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

6.2.2.11

Whatever an identifier or label denotes at its defining-point shall be denoted at all applied occurrences of that identifier or label.

NOTES

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

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

6.2.3 Activations

6.2.3.1

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

6.2.3.2

The activation of a block shall contain

a) for the statement-part of the block, an algorithm, the completion of which shall terminate the activation (see also **6.8.2.4**);

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

c) for each variable-identifier having a defining-point for the region that is the block, a variable possessing the type associated with the variable-identifier;

d) for each procedure-identifier local to the block, a procedure with the procedure-block corresponding to the procedure-identifier, and the formal-parameters of that procedure-block;

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

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

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

AVA Documents
Document Sharing Hub

Full document details: ISO IEC 07185-1990

6.2.3.3

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

a) the activation containing the procedure or function; and

b) all activations that that containing activation is within.

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

Within an activation, an applied occurrence of a label or variable-identifier, or of a procedure-identifier or function-identifier local to the block of the activation, shall denote the corresponding program-point, variable, procedure, or function, respectively, of that activation; except that the function-identifier of an assignment-statement shall, within an activation of the function denoted by that function-identifier, denote the result of that activation.

6.2.3.4

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

6.2.3.5

All variables contained by an activation, except for those listed as program-parameters, and any result of an activation, shall be totally-undefined at the commencement of that activation. The algorithm, program-points, variables, procedures, and functions, if any, shall exist until the termination of the activation.

6.3 Constant-definitions

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

```
constant-definition =  identifier '=' constant .

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

constant-identifier =  identifier .
```

The occurrence of an identifier in a constant-definition of a constant-definition-part of a block shall constitute its defining-point for the region that is the block. The constant in a constant-definition shall not contain an applied occurrence of the identifier in the constant-definition. Each applied occurrence of that identifier shall be a constant-identifier and shall denote the value denoted by the constant of the constant-definition. A constant-identifier in a constant containing an occurrence of a sign shall have been defined to denote a value of real-type or of integer-type. The required constant-identifiers shall be as specified in **6.4.2.2** and **6.7.2.2**.

6.4 Type-definitions

6.4.1 General

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

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


AVA Documents
Document Sharing Hub

Full document details: ISO IEC 07185-1990

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

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

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

Types shall be classified as simple-types, structured-types or pointer-types. The required type-identifiers and corresponding required types shall be as specified in **6.4.2.2** and **6.4.3.5**.

simple-type-identifier = type-identifier .

structured-type-identifier = type-identifier .

pointer-type-identifier = type-identifier .

type-identifier = identifier .

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

6.4.2 Simple-types

6.4.2.1 General

A simple-type shall determine an ordered set of values. A value of an ordinal-type shall have an integer ordinal number; the ordering relationship between any two such values of one type shall be the same as that between their ordinal numbers. An ordinal-type-identifier shall denote an ordinal-type. A real-type-identifier shall denote the real-type.

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

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

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

ordinal-type-identifier = type-identifier .

real-type-identifier = type-identifier .

6.4.2.2 Required simple-types

The following types shall exist

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

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

c) *Boolean-type*. The required type-identifier **Boolean** shall denote the Boolean-type. The Boolean-type shall be an ordinal-type. The values shall be the enumeration of truth values denoted by the required constant-identifiers **false** and **true**, such that **false** is the predecessor of **true**. The ordinal

AVA Documents
Document Sharing Hub

Full document details: ISO IEC 07185-1990

numbers of the truth values denoted by **false** and **true** shall be the integer values 0 and 1 respectively.

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

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

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

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

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

6.4.2.3 Enumerated-types

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

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

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

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

Examples:

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

6.4.2.4 Subrange-types

A subrange-type shall include identification of the smallest and the largest value in the subrange. The first constant of a subrange-type shall specify the smallest value, and this shall be less than or equal to the largest value, which shall be specified by the second constant of the subrange-type. Both constants shall be of the same ordinal-type, and that ordinal-type shall be designated the *host-type* of the subrange-type.

subrange-type = constant '..' constant .

Examples:

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


AVA Documents
Document Sharing Hub

Full document details: ISO IEC 07185-1990

6.4.3 Structured-types

6.4.3.1 General

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

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

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

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

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

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

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

6.4.3.2 Array-types

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

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

index-type = ordinal-type .

component-type = type-denoter .

Example 1:

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

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

NOTE — 1 Each of the following two examples thus contains different ways of expressing its array-type.

Example 2:

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


AVA Documents
Document Sharing Hub

Full document details: ISO IEC 07185-1990

Example 3:

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

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

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

NOTE — 2 A value of an array-type does not therefore exist unless all of its component-values are defined. If the component-type has c values, then it follows that the cardinality of the set of values of the array-type is c raised to the power k.

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

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

NOTE — 3 The values of a string-type possess additional properties which allow writing them to textfiles (see **6.9.3.6**) and define their use with relational-operators (see **6.7.2.5**).

6.4.3.3 Record-types

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

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

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

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

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

field-identifier = identifier .

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

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

tag-field = identifier .

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

tag-type = ordinal-type-identifier .

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

case-constant = constant .

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

The occurrence of an identifier in the identifier-list of a record-section of a fixed-part of a field-list shall constitute its defining-point as a field-identifier for the region that is the record-type closest-containing the

AVA Documents
Document Sharing Hub

Full document details: ISO IEC 07185-1990

field-list and shall associate the field-identifier with a distinct component, which shall be designated a *field*, of the record-type and of the field-list. That component shall have the type denoted by the type-denoter of the record-section.

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

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

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

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

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

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

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

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

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

The values of a variant-part shall be the distinct pairs

$$(k, X_k)$$

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

NOTES

2 If there are n values specified by the selector-type, and if the field-list of the variant associated with the i-th value has T_i values, then the cardinality of the set of values of the variant-part is $(T_1 + T_2 + \ldots + T_n)$. There is no component of a value of a variant-part corresponding to any non-active variant of the variant-part.

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

AVA Documents
Document Sharing Hub

Full document details: ISO IEC 07185-1990

Examples:

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

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

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

6.4.3.4 Set-types

A set-type shall determine the set of values that is structured as the power set of the base-type of the set-type. Thus, each value of a set-type shall be a set whose members shall be unique values of the base-type.

set-type = 'set' 'of' base-type .

base-type = ordinal-type .

NOTE — 1 Operators applicable to values of set-types are specified in **6.7.2.4**.

Examples:

```
set of char
set of (club, diamond, heart, spade)
```

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

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

6.4.3.5 File-types

NOTE — 1 A file-type describes sequences of values of the specified component-type, together with a current position in each sequence and a mode that indicates whether the sequence is being inspected or generated.

AVA Documents
Document Sharing Hub

Full document details: ISO IEC 07185-1990

file-type = 'file' 'of' component-type .

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

Examples:

```
file of real
file of vector
```

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

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

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

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

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

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

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

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

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

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

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

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

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

NOTE — 4 The two components, f.L and f.R, of a value of the file-type may be considered to represent the single sequence f.L ~f.R together with a current position in that sequence. If f.R is non-empty, then f.R.first may be considered the current component as determined by the current position; otherwise, the current position is the end-of-file position.

There shall be a file-type that is denoted by the required structured-type-identifier **text**. The structure of the type denoted by **text** shall define an additional sequence-type whose values shall be designated *lines*. A

AVA Documents
Document Sharing Hub

Full document details: ISO IEC 07185-1990

line shall be a sequence cs ~S(end-of-line), where cs is a sequence of components possessing the char-type, and *end-of-line* shall represent a special component-value. Any assertion in clause **6** that the end-of-line value is attributed to a variable other than a component of a sequence shall be construed as an assertion that the variable has attributed to it the char-type value space. If l is a line, then no component of l other than l.last shall be an end-of-line. There shall be an implementation-defined subset of the set of char-type values, designated *characters prohibited from textfiles*; the effect of causing a character in that subset to be attributed to a component of either t.L or t.R for any textfile t shall be implementation-dependent.

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

Every value t of the type denoted by **text** shall satisfy the following two rules:

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

b) If t.M = Generation, then t.L ~t.R shall be ls ~cs, where ls is a line-sequence and cs is a sequence of components possessing the char-type.

NOTE — 5 In rule b), cs may be considered, especially if it is non-empty, to be a partial line that is being generated. Such a partial line cannot occur during inspection of a file. Also, cs does not correspond to t.R, since t.R is the empty sequence if t.M = Generation.

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

NOTE — 6 All required procedures and functions applicable to a variable of type *file of char* are applicable to textfiles. Additional required procedures and functions, applicable only to textfiles, are defined in **6.6.6.5** and **6.9**.

6.4.4 Pointer-types

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

NOTE — 1 Since the nil-value is not an identifying-value, it does not identify a variable.

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

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

new-pointer-type = '↑' domain-type .

domain-type = type-identifier .

NOTE — 2 The token nil does not have a single type, but assumes a suitable pointer-type to satisfy the assignment-compatibility rules, or the compatibility rules for operators, if possible.

6.4.5 Compatible types

Types T1 and T2 shall be designated *compatible* if any of the following four statements is true:

a) T1 and T2 are the same type.

b) T1 is a subrange of T2, or T2 is a subrange of T1, or both T1 and T2 are subranges of the same host-type.

c) T1 and T2 are set-types of compatible base-types, and either both T1 and T2 are designated packed or neither T1 nor T2 is designated packed.

AVA Documents
Document Sharing Hub

Full document details: ISO IEC 07185-1990

d) T1 and T2 are string-types with the same number of components.

6.4.6 Assignment-compatibility

A value of type T2 shall be designated *assignment-compatible* with a type T1 if any of the following five statements is true:

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

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

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

d) T1 and T2 are compatible set-types, and all the members of the value of type T2 are in the closed interval specified by the base-type of T1.

e) T1 and T2 are compatible string-types.

At any place where the rule of assignment-compatibility is used

a) it shall be an error if T1 and T2 are compatible ordinal-types and the value of type T2 is not in the closed interval specified by the type T1;

b) it shall be an error if T1 and T2 are compatible set-types and any member of the value of type T2 is not in the closed interval specified by the base-type of the type T1.

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

6.4.7 Example of a type-definition-part

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

 person = ↑ persondetails;

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


Documents
Document Sharing Hub

Full document details: ISO IEC 07185-1990

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

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

6.5 Declarations and denotations of variables

6.5.1 Variable-declarations

A variable shall be an entity to which a value can be attributed (see **6.8.2.2**). Each identifier in the identifier-list of a variable-declaration shall denote a distinct variable possessing the type denoted by the type-denoter of the variable-declaration.

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

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

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

Example of a variable-declaration-part:

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

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

NOTE — Variables occurring in examples in the remainder of this International Standard should be assumed to have

AVA Documents
Document Sharing Hub

Full document details: ISO IEC 07185-1990