List of Amendments

This Language Reference Manual applies to Pascal-XT running under the operating systems BS2000 and SINIX. Every Pascal-XT compiler with a version number of the form 2.1x, regardless of the operating system, accepts exactly the function set described in this manual.

The table below lists only those sections containing technical changes.

Section	Item	new	modified	deleted
6.3.3	Syntax of variant-part corrected		x	
9.3.1	Examples with MOD operator corrected		x	
14.3	Examples of error propagation	x		
15.6	Setmax and Setmin with empty set		x	
15.11	Raise (0) - Error propagation	x		
16.1	Argument L0 with option Standard	x		

The italic type used up to now has been modified for reasons of consistency.

Contents

Preface

This Language Reference Manual for the Pascal-XT programming system contains a description of those elements of the Pascal-XT language which are common to all implementations of this system. Implementation-defined peculiarities are referred to in those sections where they are relevant.

A general knowledge of data processing and the fundamentals of Boolean algebra is sufficient to understand this Language Reference Manual.

This Language Reference Manual is based on the Pascal standard DIN 66256 (see References, [3]). Elements and characteristics which transcend this standard are indicated by a colored background.

Notes on Reading the Manual

Structure of the manual

This Language Reference Manual is intended as a reference work in which the characteristics of the language are described as precisely as possible. For learning Pascal a number of good textbooks are available (see e.g. [5]).
Chapter 2 deals with the definitions required for describing the language.
Chapters 3 to 16 describe the characteristics of the language.
Chapters 17 to 20 provide some background information and application possibilities which have either been added with Pascal-XT or which are commonly known to cause difficulties.

At the end of each section there is a list of cross-references to other passages of relevance to an understanding of the section. The appendix contains summaries in tabular form.

References to the literature in the text are given in short-title format. The complete title of each publication referred to can be found in the References section, followed by instructions for ordering manuals.

Syntax and semantics

A Pascal program consists of a finite sequence of symbols, of which some are given in the description of the language and others are formulated by the programmer himself according to certain rules. The syntax of the language prescribes which symbols or combinations of symbols may appear in this sequence and at which points. Since the syntax can be precisely specified, we have taken it as the starting point for describing the language. The syntax is described in Backus-Naur Form (see section 2.1).

The meaning of a program (its semantics) cannot, however, be deduced from the syntactic rules. Yet it makes sense to explain the semantics of the language on the basis of its syntax. For each language construct given in this manual, the corresponding rules of its syntax appear first, following which the meaning of the construct is explained verbally and with the aid of sample programs.

It will often be necessary to seek out the appropriate syntactic rules in other sections of the manual. The alphabetical syntax summary given in Appendix A.1 can be helpful in this regard.

Upper case and lower case

To make the manual easier to read, the following rules of notation have been followed when selecting upper case, lower case or mixed notation for word symbols and identifiers:

- Word symbols (keywords) → upper case throughout ("PROGRAM")
- Required identifiers → initial capital ("Integer")
- User-defined identifiers → lower case ("my_number")

Use of hyphens

Since Pascal is a programming language with worldwide distribution, the word symbols (3.2) and required identifiers (Appendices A.2, A.3) are retained in English. When other words appear in meta-identifiers (see section 2.1) the parts are separated by a hyphen.

Other meta-identifiers consisting of several words are joined by means of underscores. When these meta-identifiers are used in the text the underscores are replaced by blanks.

Prefixes in italics

As far as the syntax is concerned, a meta-identifier with a prefix printed in italics is equivalent to a meta-identifier without this prefix. For example, *type*-name and *variable*-name are identical to name. The italicized prefix is merely used to give expression to semantic characteristics. For instance, *ordinal*-constant means that a constant of the ordinal type is required in this position.

Items specified in italics appear only in the syntax rules. The prefixes are not italicized in the text.

Notes

Notes do not belong to the definition of the language. They provide additional information for better understanding or offer tips for programming.

Structure of the subject index

When an item in the index happens to be a compound term, it is listed in such a way that the main term appears first, followed by the preceding term or terms. In a few cases the parts of compound terms are equally significant (even if different in context), so that two or more entries in the index may exist under different initial letters.

Cross-references

Backus-Naur Form:	2.1
Meta-identifier:	2.1
Required identifiers:	A.2, A.3
Concepts:	17 to 20
Complete syntax:	A.1

The Pascal-XT Compiler Family

The Pascal-XT compiler family is a family of Pascal compilers for various Siemens computers, all of which accept the language set described in this manual.

This uniformity of the Pascal-XT compiler family permits problem-free porting of Pascal-XT programs from one computer to another. In particular, the software for a mainframe can be developed, for example, on a workstation. Conversely, software developed in Pascal-XT on a mainframe can be compiled and run without modification on a workstation.

The Pascal Language

Origins and Standardization

The Pascal programming language was developed by Professor Niklaus Wirth of the Swiss Federal Institute of Technology in Zurich. The first description of the language was published in 1970. In 1974 the "Pascal User Manual and Report" [4] by K. Jensen and N. Wirth appeared. This report was regarded as the definition of the language. Unfortunately, this description was imprecise in a number of points. These difficulties were not removed until Pascal was standardized.

The standardization of Pascal began in 1977 in Great Britain. In November 1983 the first international Pascal standard, ISO 7185, was adopted. In March 1984 the German Pascal standard DIN 66256 [3] appeared; though written in German, its contents were identical to those of the ISO standard. It forms the basis of the present handbook. We would especially like to thank the Springer-Verlag of Berlin/New York/Tokyo for their permission to use original sources.

Language Features

Pascal is a higher-level programming language developed by N. Wirth above all for teaching purposes in the areas of algorithm design and programming methodology. For this reason, Pascal plays an important part in the publication of algorithms, and is used in many university courses.

The essential point of departure for Pascal is its provision of problem-oriented data structures. Many other languages only recognize implicit data structures, or such structures as are offered by the available hardware, such as fixed (31) binary. Pascal on the other hand offers a consistent approach of making machine-independent data types definable by the user. Also of importance here is the facility for naming data types and then referring to these names in other data types.
In a Pascal program, data are described as sets of values. These are referred to as a data type. Examples of basic data types are:

- ranges, such as -5..99
- enumerations of values, e.g.
  (monday, tuesday, wednesday, thursday, friday)
- predefined (required) data types such as Boolean, Char, Integer, Real

In addition to these basic data types, complex data structures may be defined. The fundamental constructs are:

– Arrays, fields:	ARRAY
– Compound structures, data records:	RECORD
– Sets:	SET
– Files:	FILE

Besides these, Pointer types may be defined for processing lists and tree-structures; these open up an entire world of relationally linked data structures.

The block concept with procedures and functions permits programming problems to be solved in a well-structured way, following the method of stepwise refinement. The formal freedom of the language makes it possible to fashion clear, lucid programs.

Within the framework of the Pascal standard, it is possible to extend the language in a manner appropriate to the areas of application involved. The purpose of this is to maintain the original positive characteristics of the language while making a few specific additions in order to create an even more useful programming tool for commercial, industrial and scientific applications.

Origins and Purpose of Pascal-XT

The Pascal-XT language offers a series of extensions that go beyond the limits of the standard. The new concepts of Pascal-XT meet the demands put on modern programming languages today. They are based on the need to develop ever-larger software systems ever more rationally and in evershorter time periods. It is therefore necessary for software development to be carried out by teams of co-workers, and to make use of reusable and adaptable software modules already developed in other projects. Moreover, the demands made on modern programming languages are based on the need to reduce maintenance costs. The software to be developed must therefore be easy to understand and modify.

In this manual, extensions to the language are indicated by a colored background. A summary of all the extensions, organized by keyword, appears in Appendix A.4.

Compliance with the Pascal Standard

Pascal-XT complies with the demands of Levels 0 and 1 of DIN 66256 (see also section 16.1).

- **Extensions to DIN 66256 Pascal**

  All extensions to DIN 66256 Pascal in this manual are indicated at the appropriate places on a colored background. Appendix A.4 summarizes these extensions.

  If "Standard" is specified (see section 16.1), only standard Pascal (DIN 66256, Level 1 or Level 0) will be accepted. Extensions will be reported as errors at compile time.

- **Errors**

  All errors are described in the appropriate sections of the Language Reference Manual and are summarized in tabular form in Appendix A.5. The errors that can be detected are described in the User’s Guides [1,2].

- **Implementation-defined characteristics**

  All implementation-defined characteristics are described in the relevant sections and are summarized in tabular form in Appendix A.6. Further characteristics are specified in the User’s Guides [1], [2].

- **Implementation-dependent characteristics**

  Implementation-dependent characteristics are described in the relevant sections and are summarized in tabular form in Appendix A.6. The individual characteristics are defined in the User’s Guides [1], [2].

Cross-references

Implementation-defined:	2.2, A.6
Implementation-dependent:	2.2, A.7
Errors:	2.3, A.5
Extensions:	A.4
Control statements:	16

Definitions

Metalanguage

The metalanguage used in this manual to specify the Pascal syntax is based on the Backus-Naur Form. Table 2-1 lists the meanings of the various metasymbols.

Metasymbol	Meaning
`=`	is defined to be
`.`	end of definition
`[ x ]`	0 or 1 instance of x
`{ x }`	0, 1 or more instances of x
`x | y` or `( x | y )`	alternatively x or y
`"xyz"`	occurs in the source (terminal symbol)
meta-identifier	indicates a syntactic unit (non-terminal symbol)

Table 2-1: Metalanguage symbols

A meta-identifier is a sequence of letters and individual hyphens and/or underscores (see section 1.1), beginning and ending with a letter.

A sequence of terminal and non-terminal symbols in a syntax rule implies the concatenation of the text that they ultimately represent (i.e. after the non-terminals have been replaced in accordance with their syntax rules). In the syntax rules given in chapter 3 this concatenation is direct, i.e. no characters are allowed to intervene. In the syntax rules given in other sections of the manual concatenation takes place in accordance with the rules set out in section 3.8.

When used to verbally describe a syntax construct, the words **"of"**, **"containing"** and **"closest-containing"** have the following meanings:

- **the x of a y**

  refers to the x occurring on the right-hand side of a syntax rule defining y.

- **a y containing an x**

  refers to any y from which an x can be directly or indirectly derived using syntax rules.

- **the y closest-containing an x**

  refers to that y which contains an x but does not contain another y containing that x.

  These syntactic conventions are used to set out certain syntactic requirements and the resultant semantic description.

  Example

```
field-list     = [(fixed-part [";" variant-part] |
                                 variant-part) [";"] ].
fixed-part     = RECORD-section {";" RECORD-section}.
```

  Verbal descriptions referring to this syntax might read as follows:

  The **fixed part** of a **field list** L ...

  The **field list** containing the **RECORD section** A ...
  (A can also be contained in a nested RECORD definition)

  The **field list** closest-containing the **RECORD section** A ...

- **"...of an Integer type"**

  The phrase "...of an Integer type" is equivalent to the phrase
  "...of the type Short_Integer or the type Long_Integer or a
  subrange thereof".

- **"...of a Real type"**

  The phrase "...of a Real type" is equivalent to the phrase
  "...of the type Short_Real or the type Long_Real".

Implementation-defined and Implementation-dependent Characteristics

This manual describes the complete performance scope of the Pascal-XT programming language. The features and characteristics are described in full, and are identical within the various implementations in which Pascal-XT exists. In several points, however, there are peculiarities related to particular implementations. Wherever this is the case, it is indicated appropriately.

- **Implementation-defined characteristics:**

  An implementation-defined characteristic may be specific to a Pascal system, but is at any rate defined. The description can be found in the User's Guide of the implementation in question.

- **Implementation-dependent characteristics:**

  An implementation-dependent characteristic may be specific to a particular Pascal system, but is not necessarily defined.

By **Pascal system** (Pascal processor) we mean a complete system that accepts a Pascal source text as input, edits this source text for execution (interpreter, compiler with link editor) and is capable of executing the edited program (see section 13.3).

Implementation-defined or implementation-dependent characteristics are not allowed when Pascal-XT programs are to be ported from one Pascal system to another.

Cross-references

Implementation-defined:	A.6
Implementation-dependent:	A.7
Executing a program:	13.3

Classifying Errors

When a Pascal source text is executed, a Pascal system may detect errors. Errors are divided into two categories.

a) Errors detected at compile time

Violations of the syntax and static semantics as described in this manual are detected and reported by the Pascal-XT compiler at compile time. Programs containing violations of this sort cannot be executed since no object code is generated for them. These violations contradict the language description and are therefore not explicitly mentioned as errors.

Examples of such violations:

- ";" before ELSE in an IF statement
- Use of undeclared identifiers
- The control variable in a FOR statement is not locally defined
- Application of operators to operands with incompatible types
- The static expression in a constant definition cannot be calculated (e.g. CONST c = 2**31; causes an overflow)

b) Errors detected at execution time (runtime errors)

Runtime errors are errors that occur during program execution due to incorrect data or faulty program logic. Detection of these errors is made by the implementation, possibly in conjunction with the Check option. Potential runtime errors are explicitly mentioned in the separate sections of this Language Reference Manual and are summarized in tabular form in Appendix A.5.

Examples of such errors:

- Division by zero
- Dereferencing a NIL pointer
- Overstepping index or subrange limits
- Reading from file f when Eof (f) = true.

Cross-references

Check option:	16
Errors:	A.5

Lexical Tokens

General

The lexical tokens for constructing Pascal programs are:

- Special symbols
- Identifiers
- Directives
- Unsigned numbers
- Labels
- Character strings.

Except within character strings, the typographical representation of a letter (upper/lower case, type style, etc.) has no effect on the meaning of the program in which it occurs. Thus, in Pascal a name is always the same whether written in upper or lower case. The notation chosen for this manual is described in section 1.1.

Special Symbols

The special symbols include special characters and word symbols.

- **Special characters**

The meanings of the special characters can change depending on the context. They are described in the appropriate chapters.

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

- **Word symbols (keywords)**

Word symbols are reserved names. They are thus not permitted as identifiers. If the "Standard" option is activated (chapter 16), the word symbols highlighted below are not reserved identifiers.

```
word-symbol =
      "AND"       | "ARRAY"     | "BEGIN"     | "BODY"      |
      "CASE"      | "CONST"     | "DIV"       | "DO"        |
      "DOWNTO"    | "ELSE"      | "END"       | "ENTRY"     |
      "EXCEPTION" | "EXIT"      | "FILE"      | "FOR"       |
      "FROM"      | "FUNCTION"  | "GOTO"      | "IF"        |
      "IN"        | "INLINE"    | "LABEL"     | "MOD"       |
      "NIL"       | "NOT"       | "OF"        | "OR"        |
      "PACKAGE"   | "PACKED"    | "PROCEDURE" | "PROGRAM"   |
      "RECORD"    | "REPEAT"    | "RETURN"    | "SET"       |
      "THEN"      | "TO"        | "TYPE"      | "UNTIL"     |
      "USE"       | "VAR"       | "WHILE"     | "WITH" .
```

- **Alternate representation**

For some special symbols there are alternate representations. One or the other representation, or a mixture of both, may be used.

```
    ↑ = @ = ^    { = (*     } = *)     [ = (.     ] = .)
```

Cross-references

Options:	16
Meaning of special symbols:	A.3

Identifiers

The syntax of identifiers is as follows:

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

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".
```

Thus, the structure of identifiers is governed by the following rules:

- Identifiers consist of letters, digits and the underscore character ("_").
- The first character must be a letter.
- All characters within an identifier are significant.
- Identifiers must not be word symbols.
- Upper- and lower-case letters are equivalent.
- Identifiers may be any length (however, the end of the line places a limit on identifier length).
- Two consecutive underscore characters are not allowed.
- The last character must not be an underscore.

Required (predefined) identifiers such as Input or Integer already have a specific meaning, but may also be subsequently redefined and used otherwise by the user. Since they already possess their predefined meanings prior to the first character of the program, they can be redefined, if desired, in the declaration part of the main program.

Examples of

Valid identifiers	Invalid identifiers	
Y	5numbers	(starts with a digit)
sum	breadth-width	(- not allowed)
Sum1	no blank	(blank not allowed)
ABC_4	_sum	(underscore at the beginning)
Field_contents	A__B	(2 underscores in succession)

Cross-references

Required identifiers: A.2

Directives

The following directives are supported:

```
directive = "forward" | "c" | "cobol" | "fortran"
          | "external" | "internal".
```

Directives only occur in procedure and function declarations, and stand as a substitute for the procedure or function block. They are not required identifiers. The only directive prescribed by Standard Pascal is **Forward**.

Cross-references

Directives:	8.1, 8.2, 8.6

Numbers

The syntax of numbers is as follows:

```
unsigned_integer-number
                = digit-sequence | "#" hexadecimal-digit-sequence

unsigned_real-number
                = integer-part "." fractional-part ["e" scale-factor]
                | integer-part "e" scale-factor

digit-sequence  = digit {digit}.

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

hexadecimal-digit-sequence
                = hexadecimal-digit {hexadecimal-digit}.

hexadecimal-digit
                = digit|"a"|"b"|"c"|"d"|"e"|"f".

integer-part    = digit-sequence

fractional-part = digit-sequence

scale-factor    = ["+" | "-"] digit-sequence
```

- **Integer-type numbers**

Unsigned Integer numbers are digit sequences. They stand for values of an Integer type. Since Pascal-XT recognizes two Integer types, the numbers have the following type, depending on their value:

- **Short_Integer**, ranging from 0 to 32767
- **Long_Integer** ranging from 32768 to 2147483647
- In Standard Pascal, unsigned Integer numbers are always of type Integer.

Unsigned Integer numbers in the program text are lexical entities and their value must lie in the range from 0 to Long_Maxint (see section 5.2). Signed Integer numbers within programs are expressions (see chapter 9) and not lexical entities. For this reason, the number -2147483648 cannot be specified in the program text since 2147483648 is greater than Long_Maxint. However, this negative number may be read in using Read or Readstring (see chapter 15).

Examples of unsigned Integer numbers

```
1
100
3056
```

Entering numbers in hexadecimal form

Integers may also be specified in hexadecimal form. They are formed of the prefix character "#", the digits 0 to 9 and the letters A to F (in upper or lower case). Since a maximum of 32 bits (= 4 bytes) are available for Integer numbers, a number in hexadecimal notation may include a maximum of 8 hexadecimal characters. A hexadecimal number of 8 digits with most significant bit set (= sign bit) counts as a negative integer; all others are non-negative.

Examples of hexadecimal numbers

```
#40          has the same value as 64
#FF          has the same value as 255
#100A        has the same value as 4106
#FFFFFFFF    has the same value as -1
#7FFFFFFF    has the same value as Long_Maxint
#80000000    has the same value as Long_Minint
```

- **Real-type numbers**

An unsigned Real number stands for a value of a Real type.
When an unsigned Real number is used in a program, its type is adapted to the context, i.e. whether it is used as a value of the type Short_Real or Long_Real (see chapter 9).
The letter "e" or "E" in a number, followed by an integer (exponent), means "times 10 to the power of" ("e" and "E" are equivalent). The limits of the representable numeric range are defined by required constant identifiers (see section 5.2).

A Real-type number can be:

- An integer, followed by a scale factor, e.g. 2e10
- An integer, followed by a fractional part, e.g. 3.14
- An integer, followed by a fractional part and a scale factor, e.g. 3.14e-10

Examples of unsigned Real numbers

```
0.1
3.14159
5e-3           means  5 times 10 to the minus 3
67.32E+18      means 67.32 times 10 to the 18th
2e9            means 2 times 10 to the 9th
```

Cross-references

Integer types:	6.2.1
Real types:	6.2.2
Required constants:	5.2

Expressions:	9
Input/output:	19

Labels

```
label          = digit-sequence .

digit-sequence = digit {digit}.

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

Labels are digit sequences to be interpreted as Integer numbers. Their value must lie between 0 and 9999. Different digit sequences having the same value stand for the same label (see also chapter 4).

Labels are used to mark statements, to which it is then possible to branch by means of GOTO statements.

Examples of labels

```
0
13
4711
09999
```

Cross-references

Label declaration:	4
GOTO statement:	10.1.4
Scope rules:	12

Character Strings

```
character-string         = "'"  {string-element        } "'"
                         | "#'" {hexadecimal-digit-pair} "'".

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

apostrophe-image         = "''".

hexadecimal-digit-pair   = hexadecimal-digit hexadecimal-digit.

hexadecimal-digit        = digit|"a"|"b"|"c"|"d"|"e"|"f".
```

Character strings are sequences of tokens enclosed in apostrophes. Each token represents a value of the required type Char (see section 6.2.3). Upper case and lower case are not equivalent. Character strings containing more than one token stand for a value of a character-string type with as many components as the string has tokens. Character strings with exactly one token stand for a value of the required type Char (see section 6.3.1.3). Like other lexical tokens, character strings must not extend beyond the end of a line.

Representing apostrophes in character strings

If a string is to contain one or more apostrophes, each must be represented by two apostrophes. Each pair of apostrophes within the string then counts as a single character.

Empty string

A character string containing no tokens is called an "empty string". It is represented by two successive apostrophes. Empty strings are prohibited in Standard Pascal.

Example

```
coded:          printed:
'PASCAL'        PASCAL
''''            '
'Don''t'        Don't
''                        {empty string}
```

Specifying hexadecimals

Character strings may also be represented in hexadecimal form. This is done by prefixing the string with the character "#". In this case, there must always be an even number of hexadecimal digits between the apostrophes.

Examples (in EBCDI code)

```
#'C1D7C5'          means the same as 'APE'
#'D781A2838193'    means the same as 'Pascal'
```

It is thus possible to code non-printable characters in character strings.

The character string
#'0D0A'

contains the two ASCII characters for the functions "carriage return" and "line feed", with which, for example, if a printer is attached that takes ASCII codes, the corresponding movements of the platen and print head can be initiated.

Cross-references

Type Char:	6.2.3
Generalized string types:	6.3.2

Separating Lexical Tokens and Comments

In Pascal programs, lexical tokens are separated from each other by

- comments
- blanks (except in character strings)
- end-of-line markers.

These are called separators.

Any number of such separators may appear between two successive lexical entities or prior to the first lexical token of a program. At least one separator must separate each pair of lexical tokens containing identifiers, word symbols, unsigned numbers or labels. A separator must not occur within a lexical token (blanks in comments and character strings do not count as separators).

In most cases, the special characters also have a separating function. However, a program can be made clearer and more readable when separators are used extensively.

Comments

Any sequence of characters enclosed in braces (or their alternate representation) is taken as a comment.

A comment beginning with a dollar sign directly after the open comment brace is called a "pseudocomment". It contains control statements for the compiler (see chapter 16).

Rules

- End-of-lines markers may also occur within the sequence of characters, i.e. a comment may extend over several lines.
- No right brace (or its alternate representation) may occur within the sequence of characters, and in particular, comments within comments are not permitted.
- Alternate representations for braces:

```
"{" = "(*"

"}" = "*)"
```

Example:

```
{This is a comment}

(*This is also a comment*)

{Braces and parentheses may also be mixed*)

{Comments may extend
  over several
  lines}

'{This is not a comment}, but a character string'
```

Cross-references

Special characters:	3.2
Word symbols:	3.2
Numbers:	3.5
Labels:	3.6, 4
Control statements:	16

Label Declarations

The label declaration part has the following syntax:

```
label-declaration-part = "LABEL" label { "," label } ";" .
label                  = digit-sequence.
```

Labels are sequences of digits. They differ from each other in their integer value, which must lie in the range 0..9999. Labels are used to mark statements to which a branch may then be made by means of GOTO statements (see section 10.1.4). The use of labels is announced by their declaration. A label declared in a block must be used exactly once in the statement part of this block (main program or subprogram) to mark a statement. Furthermore, it may occur in any number of GOTO statements in this block or in nested blocks.

The scope rules for labels are described in chapter 12.

Example

```
LABEL 1, 13, 9999;
```

Notes

– 0000013, 0013, 013 and 13 are equivalent labels.

– Labels and GOTO statements should be avoided since, if used too often, they can lead to "spaghetti programs" in which the control flow is unclear and confusing. By using IF, CASE, FOR, WHILE and REPEAT statements (see chapter 10) and subprograms (see chapter 8) it is possible to create clear and lucid programs. Moreover, GOTO statements hinder automatic code optimization by the compiler.

Cross-references

Label:	3.6, 4
Block:	8.1, 8.2, 11.1, 12.1
GOTO statement:	10.1.4
Scope rules:	12

Constants

Constant Definitions

A constant definition introduces an identifier which represents a fixed value. This value can then be accessed by specifying the introduced constant-identifier.
A constant definition part has the following syntax:

```
constant-definition-part
                = "CONST" constant-definition { constant-definition } .

constant-definition
                = identifier "=" constant ";" .
constant        = static-expression .

constant-name   = [ package-identifier "." ] constant-identifier.
```

In Standard Pascal, only signed or unsigned numbers, constant-identifiers and character strings are permitted as constants. A sign prefixed to a constant-identifier is allowed only if this identifier stands for an Integer number or a Real number.

In Pascal-XT static expressions are also permitted as constants. Static expressions are expressions that can already be evaluated at compile time. The evaluation of the expression must not cause an error to be detected at compile time (see section 2.3). Section 9.2 describes when an expression is static.

The static expression in the right-hand part of the definition must not contain the identifier in the left-hand part.

If the static expression on the right-hand side is an (unqualified) set constructor, the defined constant will be regarded as not packed unless required otherwise by the context.

Each use of the identifier is referred to as a constant-name, and stands for the value assigned to the identifier in the constant definition.
In Pascal-XT a constant-name can also be formed by prefixing a package-identifier; it then refers to a constant definition within the specification of the identified package.

The scope rules for constant-identifiers are described in chapter 12.

Notes

- Seldom-used constants can also be specified in the program directly as numbers, character strings or static expressions, without previously linking them with an identifier in a constant definition. A typical example is the output of character strings in interactive applications. The disadvantage of this approach with repeatedly used constants is obvious: as soon as the value of a constant is to be changed, all literals strewn through the program must be located in order to do so. A constant definition reduces this effort to a single change in the definition.
- In Pascal-XT, NIL can also be used as a constant on the "right-hand" side of a constant definition.

Examples

```
CONST
   three      = 3;
   e          = 2.718282;
   minimum    = - three;
   on         = True;
   dot        = '.';
   title      = 'PASCAL';
                                    { specification in hexadecimal form }
   max_byte   =  #FF;
   ship       =  #'E2C8C9D7';
                                    { use of NIL }
   end_of_list= NIL;
                                    { arithmetic expressions }
   const1     = 1/2 * e;
   numbits    = 13;
   maxval     = 2 ** numbits - 1;
                                    { Boolean expression }
   test       = numbits > 12;
                                    { SET expressions }
   digits     = ['0' .. '9'];
   hexletters = ['A', 'B', 'C', 'D', 'E', 'F']
   hexdigits  = digits + hexletters;
                                    { character string expression }
   version    = '2.0';
   header     = Concat (title, version, '88/05/01');

                                    { use of constant pi }
   pi2        = numeric.pi * 2.0;   { from a package "numeric" }

TYPE                                { ARRAY aggregate }
   hextab     = array [0..15] of char;
CONST
   hexchar    = hextab ('0','1','2','3','4','5','6','7','8','9',
                        'A','B','C','D','E','F');
```

Cross-references

Integer:	3.5
Real number:	3.5
Character string:	3.7
Hexadecimal form:	3.5, 3.7
Simple type:	6.2
Generic pointer type:	6.5.2
Static expression:	9.2
Set constructor:	9.4
Aggregate:	9.5
Scope rules:	12

Required Constants

In Pascal-XT, the constant-identifiers listed in Tables 5-1 to 5-3 are required (predefined).

Identifier	Data type	Value
False	Boolean	Ord(False) = 0
True	Boolean	Ord(True) = 1

Table 5-1 Required constant-identifiers of type Boolean

Identifier	Data type	Value
Long_Maxint	Long_Integer	2147483647 = $2^{31}-1$
Long_Minint	Long_Integer	-2147483648 = -2^{31}
Short_Maxint	Short_Integer	32767 = $2^{15}-1$
Short_Minint	Short_Integer	-32768 = -2^{15}
Maxint	Integer	(1)
Minint	Integer	(2)
Long_Minreal	Long_Real	(3)
Long_Maxreal	Long_Real	(4)
Short_Minreal	Short_Real	(5)
Short_Maxreal	Short_Real	(6)
Minreal	Real	(7)
Maxreal	Real	(8)

Table 5-2 Required numeric constant-identifiers

Implementation-defined characteristics

(1) Maxint is implementation-defined to be equal to Short_Maxint or Long_Maxint

(2) Minint is implementation-defined to be equal to Short_Minint or Long_Minint

(3) Long_Minreal is the implementation-defined least positive value of the type Long_Real

(4) Long_Maxreal is the implementation-defined greatest positive value of the type Long_Real

(5) Short_Minreal is the implementation-defined least positive value of the type Short_Real

(6) Short_Maxreal is the implementation-defined greatest positive value of the type Short_Real

(7) Minreal is implementation-defined to be equal to Short_Minreal or Long_Minreal

(8) Maxreal is implementation-defined to be equal to Short_Maxreal or Long_Maxreal

Note

Remarks on the precision of real numbers can be found in section 6.2.2.

In connection with programmed exception handling (see chapter 14) the following error numbers are reserved as negative Integer constants:

Identifier	Value	Brief description
Numeric_Error	-02	arithmetic overflow
Range_Error	-03	value outside of subrange
Set_Error	-04	value outside of SET type
String_Error	-05	maximum length of a string exceeded
Index_Error	-06	value outside the index subrange
Pointer_Error	-07	dereferencing NIL
Variant_Error	-08	accessing an inactive variant
Case_Error	-09	no CASE alternative found
File_Error	-10	file in incorrect mode
Eof_Error	-11	attempt to read beyond end of file
Open_Error	-12	error in opening a file
Read_Error	-13	syntax error when reading a number
Memory_Error	-14	memory overflow
Break_Error	-15	interrupt
Elab_Error	-16	cyclic initialization dependence
System_Error	-01	other system error

Table 5-3 Required constant-identifiers for exception handling

Cross-references

Integer:	3.5
Real number:	3.5
Integer types:	6.2.1
Real types:	6.2.2
Boolean:	6.2.4
Exception handling:	14
Ord function:	15

Data Types

With the data types we meet a fundamental characteristic of the Pascal programming language. Every value and every variable has a type. The type defines the following properties:

– the set of permissible values (value range) that a data object may assume,
– the set of operations that may be applied to the object.

Types fall into three categories:

Category	Associated types	Value ranges / Examples
simple types	Integer Short_Integer Long_Integer Char Boolean enumerated types subrange types Real Short_Real Long_Real	Minint .. Maxint Short_Minint .. Short Maxint Long Minint .. Long_Maxint character from character set (False, True) ex.: (open, closed, locked) ex.: 0..255 or ’0’..’9’
structured types	ARRAY [...] OF ... RECORD ... END SET OF ... FILE OF ... Text String [...]	arrays records sets non-textfiles textfiles variable-length strings
Pointer types	↑ ...	ex.: ↑node
generic types	Pointer Any_File Any_Type	generic pointer type generic FILE type generic type

Table 6-1 Categories of types

Simple (or scalar) types cannot be subdivided further, i.e. they do not contain any components. Apart from the enumerated and subrange types, these types can be referenced by means of required type-identifiers. The simple types (except for the Real types) are referred to as ordinal types.
A structured type is described by means of the types of its components and the way it is structured. The components of the type may have simple types, Pointer types or other structured types.
The structuring is defined, for example, by means of the keywords ARRAY, RECORD etc.
With Pointer types, identified variables can be created, processed and destroyed on the heap. The heap is a dynamically expanding and contracting area of memory. Pointer types are required when processing graphs or diagrams (e.g. trees or lists).

The type approach has the following implications:

- In comparison to other programming languages, the definition of data structures in Pascal plays a substantially more important role in the approach to solving a problem.
- The increased effort during the definition phase pays for itself during the development phase through increased transparency and simpler definition of interfaces for large development teams.
- The type approach results in more rapid error detection, since many errors are detected at compile time that with other languages would not show up until runtime.
- Selecting the data access methods at compile time generally results in the generation of highly efficient program code.

Type Definitions

A type definition introduces a new identifier standing for a type. This new identifier is referred to as a type-identifier. The type may be a new type or a previously defined type-identifier. A type definition part has the following syntax:

```
type-definition-part
                = "TYPE" type-definition { type-definition } .

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

type-denoter    = type-name | new_type

new_type        = enumerated_type | subrange_type | string_type |
                  pointer_type | structured_type .

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

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

subrange-type   = ordinal-constant ".." ordinal-constant .

string-type     = string-identifier "[" string-length "]" .

string-length   = integer-constant .

structured-type
                = [ "PACKED" ] unpacked-structured-type .

unpacked-structured-type
                = ARRAY-type | RECORD-type | SET-type | FILE-type .

type-name       = [ package-identifier "." ] type-identifier .
```

A type definition introduces an identifier which stands for the type described in the type denoter.

Each use of this identifier is referred to as a type-name and stands for the type determined by the type denoter.
In Pascal-XT a type-name can also be formed by prefixing a package-identifier.
In this case, the type name refers to a type definition in the specification
of the associated package.

In the type denoter, it is possible to refer to a previously defined type by specifying the type-name. In this case, however, the type denoter must not contain an application of the type-identifier of the left-hand side of the type definition, except when this identifier is used as a domain type for a Pointer type (see section 6.4). Each new type differs from every other new type. A new type is specified explicitly, i.e. it is not addressed via a previously defined type-identifier.

The optional specification of a constant is only permitted directly following the required type-identifier "String"; it defines the maximum length of this string (see section 6.3.2.2).
The scope rules for type-identifiers are described in chapter 12.

Note

The definition of more complex types is made simpler by being able to use a type-name wherever type specification is expected. This makes programs easier to read and to modify, and permits a data type, once defined, to be used in many places in the program.

Example of a type definition part

```
TYPE
   posint     = 0..Maxint;
   number     = Integer;
   range      = 1..150;
   color      = (red, green, yellow, blue);
   punchcard  = PACKED ARRAY[1..80] OF Char;
   line       = String [80];
   person     = ↑personinfo;
   personinfo = RECORD
      name, firstname  : line;
      age              : number;
      CASE married     : Boolean OF
         True  : (wife : person);
         False : ();
      END;
   datafile   = FILE OF punchcard;
```

Cross-references

Simple type:	6.2
Structured type:	6.3
Variable string type:	6.3.2.2
Pointer type:	6.4
Domain type:	6.4
Compatibility:	6.6
Equivalence of types:	6.6.1
Scope rules:	12

Simple Types

A simple type defines an ordered set of values. If these values correspond to whole ordinal numbers on a one-to-one basis, the type is referred to as an ordinal type. The Real types, therefore, do not belong to the ordinal types.

In the following cases only ordinal types are permitted:
- as host type of a subrange type
- as index type of an ARRAY type
- as base type of a SET type
- as tag type in a RECORD type
- as runtime variable type in a FOR statement
- as case index type in a CASE statement
- as parameter type of the required functions Chr, Ord, Pred or Succ

The types Integer, Long_Integer, Short_Integer, Real, Long_Real, Short_Real, Char and Boolean are predefined, i.e. their names are required type-identifiers.

Integer, Long_Integer, Short_Integer, Char, Boolean, enumerated types and subrange types are ordinal types.

Cross-references

Subrange type:	6.2.6
ARRAY type:	6.3.1
RECORD type:	6.3.3
SET type:	6.3.4
Tag type:	6.3.3.1
CASE statement:	10.3.2
FOR statement:	10.4.3
Ordinal functions:	15.6

Integer Types

Integer types are ordinal types and represent subsets of the whole numbers. Pascal-XT distinguishes between the two types Long_Integer and Short_Integer, whose value ranges are defined by the required constants (see section 5.2):

```
Short_ Integer = Short_ Minint .. Short Maxint;
Long_Integer  = Long_Minint  .. Long Maxint;
Integer       = Minint .. Maxint;
```

Standard Pascal only recognizes one integer type: Integer. Moreover, it only guarantees that the values of the range -Maxint .. Maxint belong to the value range of the type Integer. The value represented by the required constant-identifier Minint no longer necessarily belongs to the type Integer.

Implementation-defined characteristic

> The required type identifier Integer is implementation-defined, and is identical either to the type Short_Integer to the type Long_Integer.

Arithmetic operators and relational operators can be applied to Integer-type values. These values can be assigned, read in from textfiles and strings, or output to textfiles and strings. Furthermore, there are many required subprograms whose parameters or function results possess Integer types.

Cross-references

Numbers:	3.5
Required constants:	5.2
Ordinal type:	6.2
Arithmetic operators:	9.3.1
Relational operators:	9.3.4
Assignment:	10.1.2
Required subprograms:	15
Input/output:	19

Real Types

The values of the types Long_Real, Short_Real and Real form a subset of the real numbers.
The required constant-identifiers Minreal, Maxreal, Short_Minreal, Short_Maxreal, Long_Minreal and Long_Maxreal exist for the least/greatest representable positive real number of the particular Real type involved.

Arithmetic operators and relational operators can be applied to Real-type values. These values can be assigned, read in from textfiles and strings, or output to textfiles and strings. Furthermore, there are many required subprograms whose parameters or function results possess Real types.

Implementation-defined characteristics

– Values of the types Short_Real and Long_Real are implementationdefined subsets of the real numbers.

– The required type identifier Real is implementation-defined, and is identical to the type Short_Real or Long_Real.

– The results of arithmetic real operators and functions are approximate values of the actual mathematical results. The precision of these approximations is implementation-defined.

Note

Real-type values cannot always be represented exactly on a processor. When performing arithmetic operations, the user must limit the accumulation of rounding errors by employing suitable algorithms. Relational operations between Real numbers for identity should be avoided due to this lack of precision.

Cross-references

Required constants:	5.2
Arithmetic operators:	9.3.1
Relational operators:	9.3.4
Assignment:	10.1.2
Required subprograms:	15
Input/Output:	19

The Char Type

The values of the ordinal type Char result from enumerating the characters in the character set defined for the given implementation (ASCII, EBCDIC, ISO 7-bit, etc.). Most of these values are representable (printable) characters. Some, however, are used to control peripheral devices or to implement data exchange protocols with other devices.

The character values, starting at 0 and ascending consecutively, are assigned ordinal numbers of the type Integer. The assignment is implementation-defined (see below). In each implementation, however, the following relations apply:

- The digits to ’0’ to ’9’ are numerically ordered and consecutively ascending.
- The upper-case letters ’A’ to ’Z’ are alphabetically ordered, but not necessarily consecutive.
- The lower-case letters ’a’ to ’z’ are alphabetically ordered, but not necessarily consecutive.
- The ordinal relation between any two character values is the same as that between the corresponding ordinal numbers.

Implementation-defined characteristics

- The value range of the Char type is implementation-defined.
- The assignment of ordinal numbers of type Integer to the values of type Char is implementation-defined.

Relational operators may be applied to Char-type values. These values may be assigned, read in from textfiles and strings, and output to textfiles and strings. Furthermore, there are many required subprograms whose parameters or function results are of type Char. In Pascal-XT, the Char type is also a generalized string type (see section 6.3.2).

Notes

– The representation of printable characters may differ for different terminal and printer models.

– It contradicts the standard if a Pascal program presupposes ordinal relations which are satisfied by a given character set but are not included in the standard requirements listed above. For example, a sort algorithm which assumes a consecutively ascending sequence of letters "A" to "Z" or ' ' < 'A' does not comply with the standard.

– In the EBCDIC character set there are gaps between the letters. In some cases this causes the required functions Succ and Pred to return an unexpected character.

Cross-references

Ordinal type:	6.2
Character strings:	6.3.2
Relational operators:	9.3.4
Assignment:	10.1.2
Required subprograms:	15
Input/Output:	19

The Boolean Type

The values of the ordinal type Boolean result from enumerating the truth values represented by the required constant-identifiers True and False.

Boolean = (False, True)

The required functions Ord, Pred and Succ (see section 15.6) return the following values:

```
Ord (False)  = 0
Ord (True)   = 1
Pred (True)  = False
Succ (False) = True.
```

Boolean operators and relational operators may be applied to Boolean-type values. These values may be assigned and output to textfiles and strings. The results of all relational operators and some required functions are of type Boolean. Expressions of type Boolean are used in IF, WHILE and REPEAT statements to control the flow of the program.

Cross-references

Required constants:	5.2
Ordinal type:	6.2
Expressions:	9
Boolean operators:	9.3.2
Relational operators:	9.3.4
Assignment:	10.1.2
IF statement:	10.3.1
REPEAT statement:	10.4.1
WHILE statement:	10.4.2
Scope rules:	12
Required functions:	15
Input/Output:	19

Enumerated Types

Enumerated types are ordinal types. They have the following syntax:

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

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

An enumerated type specifies an ordered set of values by listing the identifiers which are to represent those values. These identifiers may then be used as constant-identifiers (see chapter 12). The list of identifiers is enclosed in parentheses. The ordering of the values is determined by the sequence in which their identifiers are given, i.e. if x comes before y, x is less than y.
The ordinal number of each value of an enumerated type results from the mapping of all values of the type onto consecutive non-negative values of the type Integer, starting with 0. The first identifier thus has the ordinal value 0, the second 1, and so on.

All identifiers in a declaration part must be different from each other (see chapter 12). For this reason, different enumerated types are not allowed to introduce the same identifiers. The enumerations (white, red, blue) and (yellow, green, red) must therefore not occur in the same declaration part as both introduce the constant-identifier "red".

Since identifiers are not allowed to be word symbols, the enumeration (do, re, mi, fa, sol, la, ti) is prohibited as "DO" is a word symbol (see sections 3.2 and 3.3).

The scope rules for the identifiers defined in enumerations are described in chapter 12.

The required functions Pred, Succ and Ord may be applied to enumeratedtype values. Furthermore, these values may also be used in assignments.

Note

Enumerated types make it possible to work very elegantly with information that in most other programming languages can be represented only with sequences of bits.

Examples

```
TYPE
   workday   = (mon, tue, wed, thu, fri);
   suit      = (club, spade, heart, diamond);
   color     = (red, blue, yellow, green, white, violet, orange);
   shape     = (circle, ellipse, rectangle);
```

For example, the following apply:

```
   Ord (red)       = 0
   Ord (green)     = 3
   Ord (orange)    = 6
   Succ (tue)      = wed
   Pred (tue)      = mon
   thu             > mon
```

Cross-references

Identifiers:	3.3
Relational operators:	9.3.4
Assignment:	10.1.2
Scope rules:	12
Required functions:	15

Subrange Types

Subrange types are ordinal types. They have the following syntax:

```
subrange-type    = ordinal-constant ".." ordinal-constant .
```

A subrange type is a subrange of an ordinal type, which is referred to as its host type. A host type may be required (e.g. Integer) or it may refer to a previously defined enumerated type. A subrange type is defined by specifying the least and greatest values in the subrange, separated by two periods. The first constant indicates the least value. It must be less than or equal to the greatest value, which is indicated by the second constant. Both constants must be of the same ordinal type.

In Pascal-XT, the two constants may belong to different integer types (Long_Integer, Short_Integer). In this case the host type is the type Long_Integer.

The same operations may be applied to values of a subrange type as are permitted for the associated host type.

Note

Subrange types are especially useful for those tasks where values have to lie within a certain range. On the one hand, this enhances the lucidity of the program since the value range of variables is clearly stated. On the other hand, when the Check option is activated (see chapter 16), a check is also made at runtime to see whether a variable of a subrange type has accepted any impermissible values.

Examples

```
TYPE
                                { host type Integer: }
   hour        =  0 .. 24;
   range2      = -5 .. +4711;
   range3      =  Ord('0') .. Ord('9');
                                { host type color, see 6.2.5: }
   basic-color =  red .. yellow;
   mixed-color =  green .. orange;
                                { host type Char: }
   digit       = '0' .. '9';
```

Cross-references

Constants:	5.1
Ordinal types:	6.2
Equivalence of types:	6.6.1

Structured Types

Structured types have the following syntax:

```
structured_type = [ "PACKED" ] unpacked_structured_type .

unpacked_structured_type
                = ARRAY-type | RECORD-type | SET-type | FILE-type .
```

A structured type is made up of other types. The components of a structured type may possess simple types, Pointer types or other structured types. A structured type is characterized by the types of its components and the way it is structured.

Packed structured types

The word symbol PACKED may come before the word symbols ARRAY, RECORD, SET and FILE. PACKED indicates to the compiler that values of this type may be represented internally so as to save space, even if this reduces efficiency when working with PACKED-type variables or their components.

The specification of a structured type as packed refers only to the representation of that type itself, but not to any components of the structured type. An exception to this rule is the abbreviated notation of multi-dimensional ARRAY types (see section 6.3.1).

Variable String-types are packed types.

Besides the potential effect on efficiency and storage space, the following rules for packed and unpacked types should be noted:

– Values of a packed SET type cannot be linked in expressions with values of an unpacked SET type (see section 9.3).

– When variable parameters are passed (see section 8.5.2), the actual parameter must not be a component of a variable whose type is packed (except for parameters of required subprograms such as Read or New).

– With regard to the comformability of conformant arrays, the actual parameter and the conformant array schema must both be packed or both unpacked (see section 8.5.4).

– With regard to the required procedures Pack and Unpack (see section 15.8), one ARRAY parameter must be packed and the other unpacked.

Cross-references

Simple type:	6.2
Pointer type:	6.4
Variable parameters:	8.5.2
Conformant array schemas:	8.5.4
Required procedures Pack and Unpack:	15.8

ARRAY Types

An ARRAY type defines a structure consisting of a fixed number of components, all possessing the same type. An ARRAY type has the following syntax:

```
ARRAY-type     = "ARRAY" "[" index-type { "," index-type } "]"
                 "OF" component-type.

index-type     = ordinal-type-denoter .

component-type = type-denoter .
```

The value of an ARRAY type consists of an assignment of a value from the component type to each value of the index type. The index type must be an ordinal type, e.g. Char, Boolean, Integer, enumerations or subranges. The component type may be any type.

Values of an ARRAY type may be assigned as a whole. The components may be accessed by means of indexing. Furthermore, the required procedures Pack and Unpack can be used. Additional operations may be applied to values of particular ARRAY types, e.g. fixed character strings (see section 6.3.2.1).

Note

Components of a packed ARRAY cannot be passed to subprograms as variable parameters.

Examples

In the final example a RECORD type is used as a component type (see section 6.3.3).

```
TYPE
   col_array  = ARRAY [Boolean] OF (red, yellow, green, blue);
   int_array  = ARRAY [Char] OF Integer;
   char_array = ARRAY [0..255] OF Char;
   real_array = ARRAY [-10..10] OF Real;
   rec_array  = ARRAY [1..10] OF RECORD k, g : Real; END;
```

Abbreviated notation for multi-dimensional ARRAY types

In particular, the component type of an ARRAY type may also be an ARRAY type. An abbreviated form is permitted for specifying such nested ARRAY types. The following examples demonstrate different possible ways of defining the same data type.

```
ARRAY [Boolean] OF ARRAY [1..10] OF ARRAY [1..9] OF Real
ARRAY [Boolean] OF ARRAY [1..10,          1..9] OF Real
ARRAY [Boolean,           1..10] OF ARRAY [1..9] OF Real
ARRAY [Boolean,           1..10,          1...9 OF Real
```

The abbreviated manner of writing gives rise to multidimensional arrays. With the abbreviated form, the attribute PACKED refers to all abbreviated definitions. Thus the following both have the same meaning:

```
PACKED ARRAY [1..10,                  1..8] OF Boolean
PACKED ARRAY [1..10] OF PACKED ARRAY [1..8] OF Boolean
```

Typical examples of these abbreviated definitions are matrices:

```
TYPE
     matrix = ARRAY [1..100, 1..100] OF Real;
```

Cross-references

Ordinal types:	6.2
PACKED:	6.3
Character strings:	6.3.2
Indexing:	9.6.2
Assignment:	10.1.2

Generalized String Types

The term "generalized string type" encompasses the following types:

– the required type Char,
– PACKED ARRAY [1..n] OF Char with $1 < n \leq$ Short_Maxint.
– String and String [n] with $1 < n \leq$ Short_Maxint

In Standard Pascal only the types PACKED ARRAY [1..n] OF Char are referred to as string types, where n is any Integer-constant greater than 1.

In Pascal-XT, a PACKED ARRAY [1..n] OF Char is only considered a string type when $n \leq$ Short_Maxint. This is at the same time the maximum value for n in String [n].

Relational operators may be applied to values of a generalized string type. These values may also be assigned (as a whole) and output to textfiles and strings. Their components can be accessed by means of indexing.
There are several required subprograms whose parameters or function results possess generalized string types.

Cross-references

Relational operators:	9.3.4
Type Char:	6.2.3
Assignment:	10.1.2
Required subprogram:	15
Input/Output:	19

Fixed String Types

A fixed string type is a packed ARRAY type PACKED ARRAY [1..n] OF Char with a constant n, where $1 < n \leq$ Short_Maxint.

Each value in this sort of fixed string type must contain exactly n characters.

Note

A fixed string type has the properties both of an ARRAY type and of a generalized string type.

Example

```
punchcard = PACKED ARRAY [1..80] OF Char;
```

Cross-references

PACKED:	6.3

ARRAY type: 6.3.1

Variable String Types

Variable string types (also known for short as "string types") are defined by means of the required type-identifier String in accordance with the following syntax:

```
string-type  = "String" ["[" constant "]"].
```

The constant after the required type-identifier String determines the maximum length of the variable string type. The value of the constant must be of the type Short_Integer and lie in the range 1 .. Short_Maxint.
A variable string type is considered a packed type.

Implementation-defined characteristic

If no constant is specified after the type-identifier String, an implementation-defined maximum length is assumed.

Every value of type String is a character string whose current length (number of characters) is less than or equal to the maximum length of the string type. Thus, even the empty string is a permissible string type value.

The characters in the string as well as their number (the length of the string) are components of the String type. The characters in the string may be accessed by means of indexing (see section 9.6.2); the length can be interrogated using the required function Length (see section 15.3).

Relational operators may be applied to String type values. These values can be assigned (as a whole), output to textfiles and strings, and read in from textfiles and strings (as remainder of line). Numeric values may be output in printable form to a String-variable and read from it (see required procedures Readstring and Writestring, section 15.3). Additional required subprograms can be used to manipulate string type values (see section 15.3).

Note

Components of a string type value cannot be passed to subprograms as variable parameters.

Examples

```
TYPE
   stdstring = String;         { implementation-defined maximum length }

   string10  = String [10];    { maximum length 10 }

   str       = String [20000]; { maximum length 20000 }
```

Cross-references

Relational operators:	9.3.4
Indexing:	9.6.2
Assignment:	10.1.2
Required subprograms:	15.3
Input/Output:	19

RECORD Types

A RECORD type defines a structure with a fixed number of components, which may possess different types. RECORD types have the following syntax:

```
RECORD-type      = "RECORD" field-list "END" .

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

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

RECORD-section   = field-identifier-list ":" type-denoter .

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

field-identifier
                 = identifier [ "(" offset [ ":" bit-range ] ")" ] .

offset           = Integer-constant .

bit-range        = Integer-constant ".." Integer-constant .

variant-part     = "CASE" variant-selector "OF" variant { ";" variant } .

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

tag-field        = field-identifier .

tag-type         = ordinal-type-identifier .

variant          = selector-list ":" "(" field-list ")" .

selector-list    = selector {"," selector } | "ELSE".

selector         = case-constant [".." case-constant]

case-constant    = ordinal-constant.
```

Each component of a RECORD type is called a field. For each field, a RECORD type defines a field-identifier and a type. All fieldidentifiers in a RECORD type, including any variants, must be different.

If the field list of a RECORD type is empty, then the RECORD type does not possess any field and defines a single null value.

Values of a RECORD type may be assigned (as a whole). The individual components may be accessed by means of field selection (see section 9.6.3). Variables of a RECORD type may occur in WITH statements (see section 10.5).

Example

Example of a RECORD type with only one fixed part and a nested RECORD type.

```
TYPE
   person = RECORD
      name,
      firstname  : String [20];
      age        : 0..100;
      birthday   : RECORD
         year : 0..2100;
         month: 1..12;
         day  : 1..31;
         END;
      END;
```

The field "birthday" in turn has a RECORD type, whose fields are not fields of the type "person".

Note

Components of a packed RECORD cannot be passed to subprograms as variable parameters.

Cross-references

Constants:	5.1
Field selection:	9.6.3
Assignment:	10.1.2
WITH statement:	10.5

RECORD Types with Variants

In a RECORD type with variant part, it is possible to define additional fields depending on the value of another field. The variant part consists of two or more variants which are available as alternatives. The field list in a variant part likewise conforms to the general format, i.e. it may consist of a fixed part, a variant part, or both.

The number of variants in a variant part is determined by the tag type in the variant selector. The tag type must be an ordinal type, and the type of each CASE constant must be compatible with the tag type.
The final variant of a variant part may contain the word symbol "ELSE" instead of a selector list. "ELSE" is then short for all values of the tag type which do not occur in the selector list of this variant part. If all values of the tag type already appear in the selector list, an ELSE part is not allowed.

```
A selector in the form
         c1 .. cn
is short for a selector list
         c1, c2, ..., cn,
containing all values ci of the tag type, where
         c1 ≤ ci ≤ cn .
```

The values of the CASE constants in a variant part must differ pair-bypair, and the set of their values must be identical to the set of the values of the tag tape, unless an ELSE part is specified.

The variant selector may optionally contain a tag field of the tag type, i.e. the following specifications are permissible:

CASE tag-field:tag-type OF...

or

CASE tag-type OF...

- **Tag field nonexistent**

Before a field in a variant is read- or write-accessed, precisely this variant is automatically activated. If this causes a change of variant, all fields in the newly activated variant are undefined. It is therefore illegal to change variants when read-accessing a field in a variant (since an undefined value will be read; see chapter 9).

- **Tag field existent**

Assigning a value to the tag field activates that variant which is linked to the value of the tag field. With the change of variant, all fields in this variant are undefined until they are assigned values.

The tag field will always be maintained along with the RECORD, as a sort of constituent part of the fixed part, in order to be able to determine which variant is currently active. If no tag field is specified, the active variant cannot be determined.

Example of a RECORD type with a fixed and a variant part

```
TYPE
   string20 = String[20];
   person = RECORD
      firstname : string20;
      lastname  : string20;
      age       : 0..99;
      CASE married : Boolean OF
         True  : (name_of_spouse : string20);
         False : ( );
      END;
```

The tag field possesses the type Boolean. The field list following the CASE constant "False" is the so-called empty field list. Thus, for this variant there are to be no other fields at all, apart from the fields of the fixed part and the tag field.

Notes

- A RECORD type with variants makes it possible to define structures with different versions. Each version may contain different fields, and may therefore have, for instance, different storage space requirements. An implementation can "superimpose" the variants in memory (normal case), but it can also allocate them consecutive storage space.
- We recommend against defining a FILE-type field in variants. Once the variant is activated the field is undefined, and file errors occur as a result (see also note in section 6.3.3.2).
- Once a variant is made active (switched on), all fields in this variant must first be given values, since activation causes the values in the fields to be undefined. The frequent practice of giving values to the fields in one variant and having them read under a different type in another variant is illegal. There can even be implementations which do not "superimpose" variants, and therefore do not achieve the desired functions. For type conversion of this sort, Pascal-XT provides the required procedure Convert (see section 15.10).
- Illegal accesses to inactive variants can only be detected when a tag field has been specified and the Check option is activated (see section 9.6.3).

Cross-references

Check option:	9.6.3, 16.2
Convert:	15.10

RECORD Types Specifying Representation in Memory

With RECORD types, Pascal-XT permits specifications to be made regarding the way the values of the RECORD type are represented in memory. These specifications have no effect on the execution of a Pascal program. However, they are necessary if Pascal programs are to maintain interfaces to foreign systems or linkages with other languages.

The way a RECORD-type value is represented in memory is determined by the way the values of the fields are represented. For each field, the offset can be defined relative to the start of the RECORD type, and the desired bit range can be defined for values of the field.

The memory areas for different fields of the fixed part of a field list are not allowed to overlap. For fields in different variants of a field list, there may be implementation-defined restrictions (see below), especially if the variants contains fields of type FILE.

Implementation-defined characteristics

- The size of a unit of memory is implementation-defined. It may be a byte or a multiple of a byte (e.g. times a power of 2).
- Specification of offset and bit range in field identifiers of type RECORD may be subject to implementation-defined restrictions such as alignment conditions for offset specifications, alignment of fields or allocation of variants (see also notes below).

- **Offset**

For each RECORD-type value, the components corresponding to the individual fields are stored with a fixed offset from the beginning of the entire RECORD. This offset is dependent only on the RECORD type itself, not on the variables or components of this type. If no offset is specified for a RECORD field, the offset will be determined by the compiler. Specifying an offset causes all component values associated with the declared field always to be stored in accordance with the specification relative to the beginning of the storage area reserved for the entire RECORD value.

The value of a constant specified as offset must be of type Integer, and must be non-negative. The values of the offsets specified in a field list must be specified in ascending order, corresponding to the textual sequence of the field identifiers.

- **Bit range**

Bit ranges may only be specified in packed RECORD types, and only for ordinal-type or Pointer-type fields. The bit range then indicates the position of a bit sequence, relative to the memory unit specified with the offset, at which the values of the RECORD field are stored, right-justified. The numbering of the bits begins with "0" (this is the first bit within the memory unit determined with the offset), continues up to the last bit of that memory unit, and then continues with the first bit of the next memory unit (that is the memory unit specified by the value of the offset, incremented by 1), and so forth.

The constants in a bit range must be of type Integer, and must be non-negative. The first constant determines the first bit belonging to the bit sequence, the second determines the last bit. Bit ranges must always be chosen large enough to accommodate all possible values of the type of the field.

Bit ranges referring to one and the same offset value must be specified in ascending, non-overlapping order, corresponding to the textual sequence of the field identifiers.

Notes

– Storage requirements for representing values can be found in the user's guide.

– The way FILE variables are represented in memory is implementation-dependent and unknown to the user. RECORD variables containing FILE-type fields should therefore not be passed to external systems. In particular, we recommend against including offset specifications in a RECORD type of this sort.

– An implementation may "superimpose" the variants belonging to a variant part in order to optimize storage, but this is not mandatory. In particular, if variants contain FILE-type fields, a different form of implementation-dependent allocation (e.g. consecutive) may be chosen. When FILE-type fields are used, the use of offset specifications may be subject to additional implementation-defined restrictions (see User's Guide).

Example 1

In the example below, the date is specified as a RECORD type in packed form. Figure 6-1 shows how the date December 27, 1987, is represented in memory. The offset and bit range specifications refer to an implementation for which the memory unit is one byte.

```
TYPE
   date = PACKED RECORD
      day   (0: 0..4) : 1..31;
      month (0: 5..8) : 1..12;
      year  (1: 1..15): 0..3000;
      END;
```

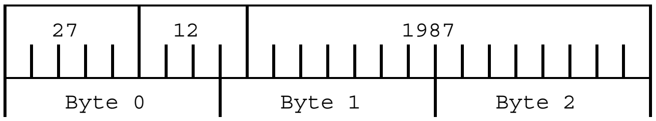


Fig. 6-1 Representation of RECORD type in memory

Example 2

Offset and bit range specifications in a RECORD type with variant part. The offset and bit range specifications may differ depending on how the implementation is defined.

```
TYPE
   t1 = PACKED RECORD
      a (0: 0..3) : 0..15;
      b (0: 4..7) : (state1, state2, state3);
      CASE day (1) : Boolean OF
         True   : (c (2) : Integer);
         False  : (d (2), e (3) : Char)
      END;
```

Cross-references

PACKED:	6.3
New, Dispose:	15.2

SET Types

In Pascal, sets are described by means of SET types. These SET types have the following syntax:

```
SET-type  = "SET" "OF" base-type .

base-type = ordinal-type-denoter .
```

The elements of a set must be values of an ordinal type, which is then called the base type of the set. The structure and values of a SET type are defined by the power set of the set of values of this base type. Thus, each value of a SET type is a set, which is either empty or consists of an arbitrary pairwise combination of different values of the base type.

There is no restriction on the least or greatest value of the base type, so long as the number of values of the base type does not exceed an implementation-defined limit; in particular, the base type may also be partly or entirely in the negative subrange of an Integer type.
In the latter instance, it is necessary to specify qualified set constructors
(see section 9.4).

Implementation-defined characteristic

The maximum number of values of a base type of a set may be restricted to an implementation-defined range.

Values of a SET type may be assigned (as a whole). Set and relational operators may be applied to them. The IN operator may be used to query whether a particular value is an element of a set. Values of a SET type are created by means of set constructors.

Examples of permissible SET types

```
SET OF 0..50
SET OF -20..20
SET OF -30000..-29990
SET OF 12321..12345
SET OF Char
SET OF Boolean
SET OF (yellow, red, blue)
SET OF ’0’..’9’
```

Values of the type SET OF (yellow, red, blue) might include:

```
[]                       {  the empty set }

[yellow], [red], [blue]  {  single-element sets }

[yellow, red],           {  two-element sets }
[yellow, blue]
[red, blue]

[yellow, red, blue]      {  three-element set }
```

Note

SET types make it possible to work very elegantly with information that in most other programming languages can be represented only with sequences of bits.

Cross-references

Ordinal types:	6.2
Set operator:	9.3.3
IN operator:	9.3.3
Relational operators:	9.3.4
Set constructor:	9.4
Assignment:	10.1.2

FILE Types

A FILE type describes sequences (theoretically of any length) of values of a specified component type, together with a current position in each sequence and a processing mode indicating whether the sequence is being read or written. The current position within the file is also referred to as the file pointer. An empty file corresponds to an empty sequence.

Pascal recognizes only sequential files. These can only be processed sequentially from beginning to end. Only one component can be accessed at a given time. To do so, a buffer variable is associated with every variable of type FILE (see chapter 7).

There are two required FILE-type identifiers: Any_File (see section 6.5.1) and Text (see section 6.3.5.2).

Permissible operations on files are implemented exclusively by accessing their buffer variables and by means of required subprograms.

Note

Pascal-XT implementations may offer a number of extensions to file processing in predefined packages, such as direct access, multiple file processing, various opening modes, and so forth.

Cross-references

Buffer variables:	7
Scope rules:	12
Required subprograms:	15
Input/Output:	19

General Files

The syntax of a FILE type is as follows:

```
FILE-type      = "FILE" "OF" component-type .

component-type = type-denoter .
```

The component type in a FILE type is not allowed to be a type denoter which itself designates a FILE type or a structured type with any (direct or indirect) FILE-type component.

Note

The use of Pointer types or of structured types possessing Pointertype components (directly or indirectly) is only meaningful for local files written and read within the same program run. Once a program run has terminated, all identifying values become invalid, so that in any subsequent program run the dereferencing of identifying values read from files will lead to errors.

Examples of FILE types

```
FILE OF ARRAY [1..50] OF Integer
FILE OF RECORD
   field  : ARRAY [1..50] OF Char;
   number : Integer;
   bool   : Boolean;
   END;
```

Cross-references

Pointer types, identifying values:	6.4

The FILE Type "Text"

The required type-identifier Text represents a special FILE type which has an additional property: it describes a sequence of lines, where each line consists of a sequence of characters of type Char. Each line is terminated by a special component value end-of-line; within a Pascal program, this value is treated as a blank (' '), except in the case of the required subprograms Reset, Readln, Eoln, Writeln and Page. A blank line consists solely of the end-of-line component.

A textfile is read only in complete lines, i.e. lines ending with an end-of-line marker. This applies even if the last line of the file was not terminated with Writeln when the file was written.

A file of type Text is referred to as a textfile. All required subprograms which process variables of type FILE OF "type" may be applied to textfiles. In addition, there are also the required subprograms

– Readln
– Writeln
– Eoln
– Page

which are only applicable to textfiles.

When textfiles are input or output, it is also possible to output character strings, Integer numbers, Real numbers and Boolean values in addition to characters. Input/Output of numbers also includes conversion of the input character string to internal representation (binary format) and vice versa. Boolean values are output according to their value as character strings (see section 15.1 and chapter 19).

Input and **Output**

The required variable-identifiers Input and Output stand for textfiles (see section 11.5).

Cross-references

Input/Output:	11.5, 15
Required subprograms:	15
File processing subprograms:	15.1

Pointer Types

Pointers are references to identified variables which can be created and destroyed during program execution. Identified variables are described in greater detail in chapter 7.

The creation and destruction of identified variables takes places only via required subprograms (see section 15.2). Chapter 20 describes how to work with pointers and identified variables.

Pointers may be assigned. However, in this case it is not the identified variable but only the reference to it which is copied. The relational operators "=" and "<>" may be applied to pointers. Here, too, it is not the values of the identified variables but only the references which are compared. Access to identified variables takes place by dereferencing the pointers.

It is possible to write reusable packages with the aid of the generic pointer type Pointer, which is described in section 6.5.2.

The syntax of Pointer types is as follows:

```
pointer-type  = "↑" domain-type .

domain-type   = type-identifier .
```

The type-identifier of the domain type can be used even before it is defined (see also private Pointer types). This enables the definition of recursive data structures (see example).

The values of a Pointer type consist of the NIL value and a set of identifying values, each of which references its own identified variable of the domain type.

The set of identifying values of a Pointer type is changed during program execution by the creation and destruction of identified variables. Each identifying value of a Pointer type, however, is only allowed to reference one identified variable with the domain type specified in the definition of the Pointer type. This strong type binding of pointers enhances the program reliability.

In Standard Pascal, the word symbol NIL does not have a special Pointer type, but adopts a suitable Pointer type so as to satisfy the rules of compatibility or assignment-compatibility. Since the NIL value is not an identifying value, it does not point to an identified variable.
In Pascal-XT, NIL is considered a value of the generic pointer type Pointer (see section 6.5.2).

Let T be any data type. Then ↑T is the Pointer type to the domain type T. If TP is a variable of Pointer type ↑T, then the values of TP are references (pointers) to identified variables of type T.

Example

Here the domain type is used prior to its definition.

```
TYPE
   person_ptr = ↑person;
   person     = RECORD
                   name:   string;
                   father: person_ptr;
                END;
```

Cross-references

Generic pointer type:	6.5.2
Pointer variable:	7
Relational operators:	9.3.4
Dereferencing:	9.6.4
Assignment:	10.1.2
Private Pointer type:	11.4
Scope rules:	12
Required subprograms:	15.2
Concept:	20

Generic Types

The Required FILE Type "Any_File"

In order to offer extensions for file processing in predefined packages, Pascal-XT has also introduced the generic FILE type. This type is known to by the required type-identifier Any_File.

The type Any_File is only permitted as a type of variable parameters, but not as the type of a variable or a component of a variable. The required file processing subprograms cannot be called with an actual parameter of this type.

Cross-references

File processing subprograms: 15.1

The Required Pointer Type "Pointer"

Additionally, Pascal-XT recognizes the generic pointer type, which is known by the required type-identifier Pointer. With the aid of this type, it is possible to write broadly applicable subprograms for processing identified variables. It thus becomes possible to write, for instance, a set of sub-programs for managing linearly concatenated lists which can be used universally for lists of different element types (see example).

The set of identifying values of the generic pointer type Pointer is the union of the identifying values of all Pointer types defined in the program. The generic pointer type is compatible with every other pointer type. The generic pointer type cancels strong type binding since a Pointer variable of this type can reference identified variables of any domain type. However, it is wrong to misuse this property for indirect type conversion (see section 6.6.2).

Data items of the generic pointer type cannot be dereferenced or used as parameters for calling the required procedures New or Dispose. Before dereferencing takes place, the identifying values must first be copied to variables of a Pointer type defined in the program.

Example

```
TYPE
   list = ↑list_element;
```

```
list_element = RECORD
   object: Pointer;
   tail  : list
   END;
```

```
PROCEDURE insert (VAR queue : list; item : Pointer);
VAR
   l : list;
BEGIN
   New (l);
   l↑.object := item;
   l↑.tail   := queue;
   queue      := l;
END;
```

The Required Type "Any_Type"

When parameters are passed to variable parameters, the actual parameter must have the same type as the formal parameter. In the case of mixed languages this strict type check can sometimes be a hindrance if, for example, addresses of variables of different types have to be passed to the same external procedure. A required generic type, identified by the required type-identifier Any_Type, makes it possible to circumvent this type check.

The type Any_Type may occur only as a type of formal variable parameters in those procedures and functions in which a directive other than Forward has been specified.

Example

In this example, variables of any type may be passed to the external procedure "ext", i.e. the addresses of the variables are available in ext. The size of each of the variables is likewise passed as a second parameter.

```
VAR
   buf1 : ARRAY [1..100] OF Integer;
   buf2 : ARRAY [1..50]  OF Real;
   buf3 : RECORD  ... END;

PROCEDURE ext (VAR buffer: Any_Type; size: Integer); external;

BEGIN
   ext (buf1, Sizeof(buf1));
   ext (buf2, Sizeof(buf2));
   ext (buf3, Sizeof(buf3));
END
```

Cross-references

Variable parameters:	8.5.2
Directives:	8.6

Equivalence and Compatibility of Data Types

Equivalence of Data Types

Equivalence of data types is required:

- when VAR parameters are passed (see section 8.5.2),
- for parameters and function results from procedural and functional parameters (see section 8.5.3),
- when there are two or more actual parameters to be assigned to formal parameters from a single conformant array parameter specification (see section 8.5.4),
- for conformability of conformant array schemas (with regard to the component types, see section 8.5.4).

According to the syntax given in section 6.1, a type denoter may possess either of the two following formats:

a) New type:
In this case, the type denoter defines a new type which is not identical to any other new type (not even when the texts of the two type denoters are completely identical).

b) Type-name:
In this case, the type designated by the type denoter is identical to the type specified in the type definition of the type-name. By using type-names as type denoters in type definitions, it becomes possible to define two or more type-names designating identical types.

Examples of a)

```
TYPE
   t1 = ARRAY [1..5] OF Char;
   t2 = ARRAY [1..5] OF Char;
```

Here, despite the completely identical notation of their type denoters, t1 and t2 indicate two different, even mutually incompatible types.

```
VAR
   i:    1..5;
   j, k: 1..5;
```

The variables i and j possess different types (which, however, being subrange types of the same host type Integer, are mutually compatible; see section 6.6.2).

The variables j and k, on the other hand, possess the same type.

Examples of b)

```
TYPE
   range    = 0 .. Maxint;
   cardinal = range;
   positive = cardinal;
   natural  = range;
VAR
   a: range;
   b: cardinal;
   c: positive;
   d: natural;
```

Here range, cardinal, positive and natural all indicate the same type. The variables a, b, c and d likewise all have the same type.

The following examples illustrate that the type-name can also be predefined or defined in a different package:

```
TYPE
   generic_ptr = Pointer;
   compl_nr    = complex_definitions.complex;
```

Here generic_ptr and the required type-identifier Pointer both indicate the same type, namely, the required generic pointer type. Similarly, compl_nr indicates the same type as the type-identifier "complex" which was defined in a package complex_definitions (referenced via a WITH clause; see example in chapter 17).

```
VAR
   x: Long_Real;
   y: Long_Real;
```

Since the type denoter Long_Real is a (required) type-name (and not a new type, as is 1..5 in the above example), the variables x and y have the same type.

Cross-references

VAR parameters:	8.5.2
Parameter subprograms:	8.5.3
Conformant array schemas:	8.5.4
Type denoter:	6.1
New type:	6.1
Type name:	6.1
Compatibility of types:	6.6.2
Subrange type, host type:	6.2.6
Pointer, generic pointer type:	6.5.2
Packages:	11.2

Compatible Data Types

Compatibility of data types is required:

- for the operands of relational operators (see section 9.3.4).
- for comformability of conformant array schemas (regarding their index types) (see section 8.5.4).
- for initial and final values in the FOR statement (see section 10.4.3)
- for assignment-compatibility (see section 6.6.2).
- for the operands of set operators (see section 9.3.3).
- for CASE constants in variant parts
- for CASE constants in CASE statements.

The compatibility requirement is checked at compile time.

Two types T1 and T2 are designated to be **compatible** if any of the following statements is true:

a) T1 and T2 are the same type (see section 6.6.1).

b) T1 is a subrange of T2, or T2 is a subrange of T1, or T1 and T2 are subranges of the same host type T3.

c) T1 and T2 are SET types with compatible base types and are either both packed or both unpacked.

d) T1 and T2 are fixed string types, where the number of components of T1 and T2 may differ. In Standard Pascal, T1 and T2 must contain the same number of components.

e) T1 (T2) is a variable string type and T2 (T1) is a generalized string type (Char type, fixed string type, variable string type).

f) T1 (T2) is a Pointer type and T2 (T1) is the generic pointer type.

g) T1 (T2) is the type Short_Integer and T2 (T1) is the type Long_Integer.

h) T1 (T2) is the type Short_Real and T2 (T1) is the type Long_Real.

Cross-references

Integer type:	6.2.1
Real type:	6.2.2
Subrange type, host type:	6.2.6
Generalized string type:	6.3.2
Fixed string type:	6.3.2.1
Variable string type:	6.3.2.2
Variant part:	6.3.3.1
SET type, base type:	6.3.4
Pointer type:	6.4
Conformant array schemas:	8.5.4
Set operation:	9.3.3
Relational operator:	9.3.4
CASE statement:	10.3.2
FOR statement:	10.4.3

Assignment-Compatibility of Data Types

When a value is assigned to a variable, the type of the value must be assignment-compatible with the type of the variable. Assignment compatibility is required if an assignment takes place explicitly (cf. items 1 to 3) or implicitly (cf. items 4 to 9):

1) When a value is assigned to a variable (see section 10.1.2).
2) When a value is assigned to a function identifier (see section 8.5.3).
3) For initial and final values in the FOR statement (see section 10.4.3).
4) When value parameters are passed (see section 8.5.1).
5) For aggregates (see section 9.5).
6) For Read (f, v) (see chapter 15).
7) For Write (f, a) (see chapter 15).
8) For indexing (see section 9.6.2).
9) For set constructors (see section 9.4).

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

a) T1 and T2 are the same type and this type is neither a FILE type nor does it contain (directly or indirectly) a FILE-type component.

b) T1 is a Real type and T2 an Integer type.

c) T1 and T2 are compatible ordinal types and the value of type T2 lies in the value range of type T1.
When a program is executed a Range_Error, an Index_Error or a Set_Error will occur if this condition is not met.

d) T1 is the type Short_Real and T2 is the type Long_Real and an approximation of the value of type T2 lies in the value range of type T1.
When a program is executed a Numeric_Error will occur if this condition is not met.

e) T1 and T2 are compatible SET types and all elements of the value of type T2 lie in the value range of the base type of T1.
When a program is executed a Set_Error will occur if this condition is not met.

f) T1 and T2 are compatible fixed string types with the same number of components.

g) T1 is a variable string type with maximum length n and T2 is a generalized string type with up to n components (see Table 6-1).
When a program is executed a String_Error will occur if T2 contains more than n components.

h) T1 is a fixed string type with n components and T2 is a variable string type. The actual length of the character string of type T2 must be exactly n.
When a program is executed a String_Error will occur if the actual length of the character string of type T2 is not equal to n.

i) T1 and T2 are compatible Pointer types.
When a program is executed an error (with unpredictable results) will occur if T2 is the generic pointer type and the value of T2 is an identifying value pointing to an identified variable whose type does not match the domain type of T1.

v := e		
Type of v	Type of e	assignment-compatible?
Char	Char PACKED ARRAY [1..k] OF Char String[n]	yes no no

PACKED ARRAY [1..m] OF Char	Char
	
	PACKED ARRAY [1..k] OF Char
	String[n]
		no
		
		if m = k
		
		if m = Length(e)

String[n]	Char
	PACKED ARRAY [1..k] OF Char
	String[k]
		yes
		if n >= k
		
		if n >= Length(e)

Table 6-1 Assignment-compatibility for character strings

Notes

– In Pascal-XT, a PACKED ARRAY [1..n] OF Char is not a generalized string type unless 1 < n <= 32767 (see section 6.3.2).

– Which of the aforementioned errors is detected during program execution is implementation-defined.

– When a value of a variable string type is assigned to a variable of a generalized string type with n components, the actual length of the variable string must be exactly equal to n. If the variable string has a length other than n, it is possible with the aid of a user-defined function to set the string to the desired actual length (e.g. by truncating or padding with blanks). The definition of the function might look as follows:

```
CONST
   std_str_length = 80;
TYPE
   std_string     = String[std_str_length];

FUNCTION adjust (s: std_string; len: Short_Integer): std_string;
TYPE
   blanks = PACKED ARRAY [1 .. std_str_length] OF Char;
CONST
   filler = blanks (’ ’: std_str_length);
BEGIN
   IF Length(s) >= len THEN
      adjust := Substring (s, 1, len)
   ELSE
      adjust := Concat
                   (s, substring (filler, 1, len - Length(s)))
END;
```

– When an Integer-type expression is assigned to a Real variable the (Integer) value of the expression is converted to a Real value prior to the assignment. However, the expression is computed in integer arithmetic (see also section 9.3.1).

Examples

```
VAR  range      : 1 .. 5;
     string5    : String [5];
     fixstr3    : PACKED ARRAY [1..3] OF Char;
     set        : SET OF 0 .. 255;
     short      : Short_Real;


 { Examples in which assignment-compatibility    }
 { is violated during program execution          }

BEGIN
     range      := 6;             { causes a Range_Error }

     string5    := ’123456’;      { causes String_Error }

     string5    := ’12’;
     fixstr3    := string5;       { causes a String_Error }

     set        := [1, 256];      { causes a Set_Error }

     short      := 10E100         { causes a Numeric_Error }
END;
```

Cross-references

Ordinal type:	6.2
Integer type:	6.2.1
Real type:	6.2.2
Component:	6.3
Generalized string type:	6.3.2
Variable string type:	6.3.2.2
SET type:	6.3.4
FILE type:	6.3.5
Pointer type:	6.4
Function:	8.2
Value parameter:	8.5.1
Set constructor:	9.4
Aggregate:	9.5
Indexing:	9.6.2
Assignment:	10.1.2
Read, Write:	15.1
Length:	15.3

Attributes of Data Types

Besides the value range that defines them and their permissible operators or functions, data types also have a number of attributes which, for example, affect how the values of the types are represented. The values of these attributes can be obtained via the various required functions (see section 15.9).

Alignment

Values of a type may be subject to implementation-dependent alignment in memory. This alignment is a multiple of the implementation-defined memory unit (see section 6.3.3.2). The function Alignof provides the requested alignment.

Storage requirements in memory units

An implementation-dependent number of memory units is required in order to represent the value of a type. The function Sizeof returns the number of memory units occupied.

Storage requirement in bits

An implementation-defined minimum number of bits is required in order to represent an ordinal-type value. The function Bitsizeof returns the minimum number of bits required.

Least value of an ordinal type

The least value of an ordinal type is obtained by means of the function function First.

Greatest value of an ordinal type

The greatest value of an ordinal type is obtained by means of the function Last.

Maximum length of a variable string type

The values of a variable string type are strings of variable length. The maximum permissible length of a character string of a variable string type is limited by the maximum String-type length specified in the type definition. The function Maxlength returns the maximum length of a variable string type.

Cross-references

Attribute functions:	15.9

Variables

A variable is a place holder for values. It may be assigned various values during program execution (see section 10.1.2). Variables may also occur as operands in expressions; they then represent the value most recently assigned to them (see chapter 9). In Pascal, every variable has a type. A variable may only be assigned values of its own type.

Variables can be categorized by type of access:

Entire variables	declared variables which are referenced in their entirety.
Component variables	components of variables with a structured type (ARRAY type, RECORD type, String type). The components are likewise variables and may be individually referenced.
Identified variables	variables pointing to an identifying value (see also section 7.2).
Buffer variables	variables which are linked to variables of type FILE (see also section 7.2).

All these categories of variables are subsumed under the term variable access (variable-object). Section 9.6 describes in detail how to access variables.

Cross-references

Expression:	9
Assignment:	10.1.2

Variable Declaration

Variable declarations have the following syntax:

```
variable-declaration-part
                = "VAR" variable-declaration
                       {variable-declaration}.

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

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

variable-name   = [ package-identifier "." ] identifier.
```

Each identifier in the identifier list of a variable declaration stands for a single variable whose type is determined by the type denoter in the variable declaration. A variable is only allowed to assume values of the specified type.

Each use of this identifier is a variable-name which stands for this variable.
A variable-name may also be formed by prefixing a package-identifier; it then references a variable declared in the specification of the indicated package.

Values of simple or Pointer variables cannot be broken down any further. Values of structured variables are made up of the values of their components.

Example of a variable declaration part

```
VAR
   a, b, c : Char;
   x, y    : Integer;
   i       : -10..+10;
   bool    : Boolean;
   field1  : ARRAY [1..10, 1..100] OF Real;
   p1, p2  : person;   { person was defined in section 6.3.3 }
```

Cross-references

Identifier:	3.3
Type denoter:	6.1
Simple type:	6.2
Structured type:	6.3
Pointer type:	6.4
Variable declaration part:	11.1
Packages:	11.2
Scope rules:	12

Categories of Variables

Variables are declared (static variables), created dynamically (identified variables) or declared implicitly (buffer variables).

Declared variables

Declared variables are declared in a variable declaration part and may be referenced via their identifier. They exist during execution of that block (program, procedure or function) where they were declared (for further information see section 13.3). Variables declared directly in a package specification, a package body or a main program exist for the entire duration of the program run.

Identified variables

Identified variables are not declared in a variable declaration part and cannot be referenced directly via an identifier. Instead, they are created by the required procedure New when a program is being executed (see section 15.2). They exist so long as there are identifying values referring to them, but not beyond the end of the program. Identifying values may also be destroyed by means of the required procedure Dispose (see section 15.2). Identified variables are accessed by dereferencing a Pointer object (variable, constant, function result).

Buffer variables

Declaring a variable f of type FILE (known for short as a *FILE* variable) implicitly defines a buffer variable which is referenced by means of f↑. The type of this variable is the same as the component type of the FILE type. With textfiles, the component is the required type Char. The buffer variable is required for accessing the components of of the *FILE* variable.

Cross-references

Textfile:	6.3.5.2
FILE type:	6.3.5
Pointer type:	6.4
Domain type:	6.4
Object:	9.6
Variable access:	9.6
Block:	12.1
Program execution:	13.3
Buffer variable:	6.3.5, 15.1, 19

Defined and Undefined Values of Variables

When used in an expression, variables must have a valid value belonging to their type. The items below describe when variables are defined, i.e. when they possess a defined value.

At the moment of its creation a variable is totally undefined. A declared variable is created when the block where is it directly defined is executed, and destroyed when this block is terminated (see chapter 12). An identified variable is created by means of the required procedure New.

- **Simple variables**

The value of a simple variable is defined only if the variable has been assigned a value; otherwise, it is undefined. A variable may be assigned a value by means of assignment (see section 10.1.2).

- **Structured variables**

The value of a structured variable is either defined (in which case all components possess a valid value) or totally undefined (in which case none of its components has a valid value) or undefined (in which case not all of its components have a valid value). Components of an undefined structured variable may be used in expressions insofar as these components have a defined value. The entire structured variable cannot be used in expressions until all of its components have defined values.

- **Pointer variables**

A Pointer variable is defined if it has a valid identifying value or the value NIL. It may be assigned a valid identifying value by calling the required procedure New (see section 15.2) or by assignment (see section 10.1.2).
Identifying values are destroyed by calling the required procedures Dispose and Release (see section 15.2).

Once New has been called, the specified Pointer-variable has a valid value, namely, a reference to the created identified variable. This created identified variable, on the other hand, is totally undefined.

- **FILE variables**

A FILE variable has a defined value if one of the following propositions is true:

1) It was defined by means of the required procedure Rewrite.

2) The variable has been assigned a file existing outside the program. This assignment may be made by means of the required procedure Assignfile (see section 15.1) of by specifying the variable as a program parameter (see section 11.5).

Accordingly, a FILE variable is undefined if it was not defined with Rewrite or if there is no assignment to a file existing outside the program.

Notes

– Non-initialized variables are a frequent source of errors which only become apparent at a later point in time during program execution as a result of random consequent errors (e.g. overwriting of code and data). The effect of executing such errored programs is unpredictable and may vary depending on other factors (load address, link sequence, and so forth).

– In some cases the compiler option Initialize, together with the Check option, can help in the detection of non-initialized variables. When variables are created (see above) the storage areas for the variables are preset with a particular "bit pattern". If this bit pattern represents an invalid value of the variable, a runtime error will occur in many cases when the variable is used.
The Initialize option, however, does not cause variables to be set with valid initial values (see chapter 16).

Cross-references

Runtime errors:	2.3, A.5
Structured types:	6.3
Simple type:	6.2
Component variable:	6.3
Program parameter:	11.5
Assignfile:	15.1
Rewrite:	15.1
New, Dispose:	15.2
Buffer variables:	7.2, 19
Compiler options:	16
Identified variables:	7.2, 20

Procedures and Functions

Procedures and functions are also referred to below by the collective term "subprograms".

Procedures are used to structure a program clearly and unambiguously into substeps (the principle of "stepwise refinement"). The invocation of a procedure (i.e. the execution of the statement sequence contained within it) takes place by means of a procedure call (also known as a procedure statement; see section 10.1.3).

Functions are used to clarify and simplify expressions by having elaborate or complicated interim steps written as functions. Unlike a procedure, a function has the property of representing a value within an expression. This value is formed by means of an assignment to the result variable in the statement part of the function. The identifier of this result variable is identical to that of the function.
A function call (also known as a function designator) is always an expression or part of an expression (see chapter 9).

Subprograms may also have their own declarations. The identifiers used within them are "local", i.e. they are only valid within the subprogram block where they were declared, regardless of whether outside the block (globally) there already exist like-named identifiers, possibly with a completely different meaning. Subprograms themselves may also contain further subprograms, i.e. they may be nested to any depth.

Subprograms can be repeatedly activated. This can occur from outside the subprogram's own statement sequence or even from within the subprogram (recursion). Recursion is limited in that each call requires new memory space, so that there may not be sufficient memory available.

The following (infinite) recursion always results in memory overflow (we do not advise trying it out!):

```
FUNCTION again_and_again (i:Integer) : Integer;
BEGIN
   again_and_again := again_and_again (i)
END
```

Subprograms may access the following entities:

– the subprogram’s own parameters,
– local variables and constants (defining point lies within subprogram’s declaration part),
– global variables, constants and parameters (defining point lies outside the subprogram’s declaration part).

Parameters may be used to pass values to subprograms or to return them to the calling part of the subprogram.

We highly recommend that the defining points of the variables used be placed as close as possible to the "location" at which they are processed, i.e. to use as few global variables as possible. Obviously, it will not be possible to dispense with global variables altogether; yet the user should be aware that the use of global data increases the probability of errors, side effects and so forth. Equally important, global data frequently forces the compiler to generate elaborate and hence slower code in order to access this data.

Besides the subprograms declared by the user, there is also a series of required subprograms, some of which are prescribed by the Standard while others are extensions in Pascal-XT. They are described in detail in chapter 15.

Cross-references

Errors:	2.3, A.5
Side effects:	8.2
Expression:	9
Defining point:	12.2
Required subprograms:	15

Procedure Declaration

Procedure declarations have the following syntax:

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

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

procedure-identification
                = "PROCEDURE" procedure-identifier
                  [formal-parameter-list].

INLINE-procedure-declaration
                = "INLINE" procedure-heading ";" procedure-block ";".

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

statement-part  = compound-statement.

procedure-name  = [ package-identifier "." ] identifier.
```

A procedure declaration introduces an identifier which is linked with the procedure-block. Each use of this identifier is a procedure-name, and stands for the execution of the statements contained in the procedure-block (except in the case of procedure identification; see section 8.7).
A procedure-name can also be formed by prefixing a package-identifier; it then references a procedure declaration in the identified package specification.

"Block" stands for a declaration part followed by a statement part. A statement part is a compound statement.

"Formal parameter list" is a list of formal parameters, enclosed in parentheses (see section 8.5), which serve as place holders for the actual parameters used in the procedure call.

In Pascal-XT, as an extension to the Standard, it is permitted to repeat the formal parameter list from the associated procedure declaration within a procedure identification. In this case, however, the parameter lists must be textually identical.

INLINE procedures are described in section 8.3, directives and procedure identifications in section 8.6.

Example of a procedure declaration

```
PROCEDURE swap (VAR x, y: Integer);

VAR
   temp : Integer;

BEGIN
   temp := x;
   x := y;
   y := temp;
END {swap};
```

Cross-references

Label declaration part:	4
Constant definition part:	5
Type definition part:	6
Variable declaration part:	7
INLINE subprogram:	8.3
Parameters:	8.5
Directives, procedure identification:	8.6
Compound statement:	10.2
Block:	12.1

Function Declaration

Functions represent a value: their result. They may therefore be used in expressions. The type of the result must be specified in a function declaration as a type-name following the function heading.

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

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

result-type     = type-name.

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

INLINE-function-declaration
                = "INLINE" function-heading ";" function-block ";".
function-block  = block .

function-name   = [package-identifier "." ] identifier .
```

A function declaration introduces an identifier which is linked with the function-block. Each use of this identifier is a function-name, and stands for the execution of the statements contained in the function-block (except in the case of function identification; see section 8.7).
A function-name can also be formed by prefixing a package-identifier; it then references a function declaration in the identified package specification.

Functions must include an assignment to the function-identifier, which at that point serves as a place holder for the function value. This assignment may be in the statement part of the function or in a subprogram declared within the function. When the function-block of the function is processed, at least one such assignment must be performed. In this way the result of the function is determined. If two or more such assignments are performed, the function result is determined by the assignment which was the last to be dynamically executed.

In Standard Pascal, only simple types and Pointer types are permitted as a result type.

In Pascal-XT, the result type may be any type that is not a FILE type and does not have a FILE type (directly or indirectly) as a component type. In the case of structured result types, the function identifier must be assigned a value as a whole; assignments to single components are only possible with aggregates (see example 2).

"Formal parameter list" is a list of formal parameters enclosed in parentheses (see section 8.5) which serve as place holders for the actual parameters used in the procedure call.

In Pascal-XT, as an extension to the Standard, it is permitted to repeat the formal parameter list and the result type from the associated function declaration. In this case, however, the parameter lists must be textually identical.

INLINE functions are described in section 8.3, directives and function identification in section 8.6.

Note

The task of a function is to calculate a value. It enhances the clarity and readability of programs and facilitates the writing of correct programs when the function result is calculated solely from the values of the parameters. We recommend against including the values of global variables. Even more dangerous and confusing are functions with side effects, i.e. those which also change global or identified variables after calculating the function result.

Example 1

The function "max" returns the maximum value of the two parameters.

```
FUNCTION max (a, b: Integer): Integer;
BEGIN
  IF a <= b THEN
     max := b
  ELSE
     max := a
END { max };
```

Example 2

The two functions "add" and "sub" return values of a RECORD type (structured type). The function add uses the temporary variable temp to calculate the result; the function sub uses an aggregate (see section 9.5) since component-by-component assignments in the form add.re := x.re + y.re are prohibited.

```
TYPE
   complex = RECORD
                re,
                im: real
             END;

FUNCTION add (x, y: complex): complex;
VAR
   temp: complex;
BEGIN
   temp.re := x.re + y.re;
   temp.im := x.im + y.im;
   add     := temp;
END { add };

FUNCTION sub (x, y: complex): complex;
BEGIN
   sub := complex (x.re - y.re, x.im - y.im);
END { sub };
```

Example 3

The function "replace" returns a character string. The first occurrence of the string "old" in "original" is to be replaced by "new". If old does not occur, original will be returned.

```
FUNCTION replace (original, old, new: String): String;
VAR
   i: Short_Integer;
BEGIN
   i := Position (old, original);
   IF i > 0 THEN
      replace :=
         Concat
            (Substring (original, 1, i-1),
             new,
             Substring
                (original,
                 i + Length (old),
                 Length (original) - Length (old) - i + 1))
   ELSE
          replace := original;
END { replace };
```

Cross-references

Simple type:	6.2
FILE type:	6.3.5
Global variable:	7
Identified variables:	7
INLINE subprograms:	8.3
Parameters:	8.5
Aggregates:	9.5
Assignment:	10.1.2
Block:	12.1
String functions:	15.3

INLINE Subprograms

A subprogram can be declared an INLINE subprogram by specifying the keyword INLINE before writing PROCEDURE or FUNCTION.
In this case, the subprogram block must not be replaced by a directive (see section 8.6).
At every place where an INLINE subprogram is called, the call is replaced by a copy of the subprogram block. At the same time, the formal parameters are replaced by the actual parameters in such a way that the subprogram run has the same effect as a "genuine" subprogram call.
This so-called INLINE expansion serves to optimize subprogram calls at execution time; this produces more efficient code, even if at times the code may be somewhat longer.

Within an INLINE subprogram, it is not permitted to declare non-INLINE subprograms; moreover, the block of an INLINE subprogram must not contain a recursive call of its own subprogram.

It is not permitted to declare FILE variables in an INLINE subprogram.

Section 11.2 discusses the peculiarities of INLINE subprograms in package specifications.

Example

```
INLINE FUNCTION max (a, b: Integer): Integer;
BEGIN
  IF a <= b THEN
     max := b
  ELSE
     max := a
END { max };
```

Cross-references

Directives:	8.6
Subprogram call:	8.7
Block:	12.1
Package specification:	11.2

ENTRY Subprograms

In package specifications, procedures and functions may be declared ENTRY subprograms by specifying the keyword ENTRY before writing PROCEDURE or FUNCTION. Subprograms declared in this manner may also be called by program parts which are not written in Pascal. When this is done, any implementation-defined interfaces must be maintained.

ENTRY subprograms are described in greater detail in section 11.2.

Cross-references

Package specification:	11.2

Parameters

Subprograms communicate with their environment via parameters and via global variables and constants (where "global" means declared outside the subprogram). Parameters must be declared in the formal parameter list of the subprogram declaration. They are replaced with actual parameters when the subprogram is called.

There are five types of formal parameters:

- Value parameters
- Variable parameters (VAR parameters)
- Procedural parameters
- Functional parameters
- Conformant array parameters

Formal parameter lists have the following syntax:

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

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

A formal parameter list consists of individual formal parameter sections, separated by semicolons. In each formal parameter section there is either exactly one procedural or functional parameter (see section 8.5.3), or there is a declaration of one or more values or variable parameters of the same type, or of one or more conformant array schemas.

The conformant arrays satisfy level 1 of the Standard (see section 2.2) and are dealt with in section 8.5.4.

Note

In Pascal-XT the formal parameter list may be repeated in the identification of a subprogram. However, it must be textually identical to the one given in the declaration.
With functional identifications, the parameter list, if present, and the function result must be both repeated or both omitted.

Value Parameters

Value parameters are defined by:

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

Value parameters are used for passing values to subprograms. Within a subprogram, they have the properties of local variables, to which the actual values are assigned before the subprogram block is entered.

It is not possible to use value parameters in order to have the subprogram modify the values of variables which were passed as actual parameters. In other words, value parameters cannot be used to send values outside the subprogram.

When a subprogram is called, an expression must be specified for each formal value parameter (see chapter 9). This may be, for instance, a variable, a constant or a function result, and may contain operations such as "+" or "<". The value of the expression must be assignment-compatible (see section 6.6.3) with the type of the associated formal parameter. When value parameters are passed, the following runtime errors may occur when the program is executed.

Possible runtime errors:

Numeric_Error	- At the time a value parameter is passed, the value of the actual parameter of type Long_Real does not lie in the value range of the formal parameter of type Short_Real.
Range_Error	- At the time a value parameter is passed, the value of the actual parameter of an ordinal type does not lie in the value range of the type of the formal parameter.
Set_Error	- At the time a value parameter is passed, the value of the actual parameter of type SET does not lie in the value range of the type of the formal parameter.
String_Error	- At the time a value parameter is passed, the actual length of the character string of the actual parameter is greater than the maximum length of the variable string type of the formal parameter.
	- At the time a value parameter is passed, the actual length of the character string of the actual para-meter is not equal to the length of the fixed string type of the formal parameter.
unpredictable effects	- At the time a value parameter is passed, the type of the actual parameter of generic pointer type and the identifying value of the expression point to an ident-ified variable whose type differs from the domain type of the type of formal parameter.

Further runtime errors may occur when evaluating an expression which is received as an actual parameter (see chapter 9).

Example of a procedure with value parameters and how to call it

```
{ Let i, j, z be "global" variables, f a function of type Integer or
a subrange thereof. }

   PROCEDURE totalout (x, y: Integer; b: Boolean);
   BEGIN
      IF b THEN
         Writeln (x + y);
   END;


   BEGIN
     ...
     totalout (i * j, f(z), True);
     ...
   END.
```

Cross-references

Runtime errors:	2.3, 14, A.5
Assignment compatibility:	6.6.3
Variable:	9.6
Expression:	9

Variable Parameters

Variable parameters are defined by:

```
variable-parameter-specification
                = "VAR" identifier-list ":" type-name.
```

For the period when the subprogram is being executed, a variable parameter is identified with the variable access that was specified as an actual parameter in the subprogram call. This causes an operation to be carried out on the variable access which was specified as the actual parameter.

When the subprogram is called, an actual parameter must be passed for each formal variable parameter. This actual parameter must satisfy the following conditions:

a) The actual parameter must be a variable object (variable access).

b) The actual parameter must have the same type (see section 6.6.1) as the associated formal parameter.
If, however, the type of a formal parameter is the generic pointer type (Pointer) or the generic FILE type (Any_File), then the corresponding actual parameter may be of any Pointer or FILE type.
If the type of the formal parameter is the generic type Any_Type, then the corresponding actual parameter may be of any type.

c) The actual parameter must not be a tag field of a variant part of a RECORD type (see section 6.3.3).

d) The actual parameter must not be a component of a packed-type variable (see section 6.3).

Notes

– Variable string types are likewise considered packed types. Thus, it is not permitted to pass the components of String variables as actual variable parameters.

– If the variable parameter specification, it is necessary to specify a type-name. The required type-identifier String, followed by a length specification (e.g. String [4]), is not a type-name but rather a type denoter.

– It the actual parameter is a dereferenced or an indexed object (see section 9.6), dereferencing or indexing is already performed when the subprogram is called, i.e. long before the statement part of the subprogram block is executed. If the value of the Pointer or index expression is changed while the subprogram is being executed, this does not alter the binding of the formal parameter to the variable access determined when the subprogram was called.

Example of a procedure with variable parameters and how to call it

```
VAR
   i, j: Integer;

PROCEDURE swap (VAR x, y : Integer);
VAR
  temp : Integer;
BEGIN
  temp := x;
  x    := y;
  y    := temp;
END;

BEGIN
  i := 5;
  j := 8;
  swap (i, j);
  Writeln(i);                { now yields 8 }
  Writeln(j);                { now yields 5 }
END {swap}
```

Examples of invalid actual parameters

```
TYPE
  string4 = String [4];
VAR
  a : String [4];

PROCEDURE parametertest (VAR par : string4);

   BEGIN ... END;

BEGIN
  a := '123';
  parametertest (a);        { different types }
  parametertest ('abcd');   { no variable }
END {parametertest}
```

Both parameter test calls are semantically errored. Variable a does not have the same type as the VAR parameter (see section 6.6.1), and 'abcd' is a constant and not a variable access.

Cross-references

Runtime errors:	2.3, 14, A.5
Packed types:	6.3
Variable string types:	6.3.2.2
Tag field:	6.3.3
Generic FILE type:	6.5.1
Generic pointer type:	6.5.2
Type identity:	6.6.1
Objects:	7
Variables:	7.1, 9.6
Expression:	9

Procedural and Functional Parameters

When subprograms are used as parameters they are defined as follows:

```
functional-parameter-specification = function-heading.

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

Procedural and functional parameters are specified in the form of a complete procedure or function heading. They appear as a formal parameter in the formal parameter list of another procedure or function.

When that other procedure or function is called, the corresponding actual parameter must be a procedure-name or function-name that satisfies the following conditions:

a) It must not designate a required subprogram.

b) It must not designate an INLINE subprogram.

c) It must not designate a subprogram with one of the directives c, cobol, fortran, internal or external.

d) If the formal parameter itself contains a formal parameter list, the actual parameter must also have a formal parameter list and both must match (see below).

e) If the formal parameter is a functional parameter, the actual parameter must also designate a function with the same result type as that of the formal parameter. If, however, the result type of the formal parameter is the generic pointer type, then the result type of the actual parameter may be any Pointer type.

Equivalence of formal parameter lists

The formal parameter lists of a formal parameter and an actual parameter are equivalent when they have the same number of formal parameter sections and the following propositions hold true for any two sections at corresponding positions:

a) They are both value parameter specifications or variable parameter specifications with the same number of parameters and the same type. If, however, the type in the formal parameter list of the formal parameter is the generic pointer type (Pointer), the type in the formal parameter list of the actual parameter may be any Pointer type.

b) They are both procedural parameter specifications with matching formal parameter lists.

c) They are both functional parameter specifications with matching formal parameter lists and the same result type. If the result type in the formal parameter list of the formal parameter is the generic pointer type (Pointer), the result type in the formal parameter list of the actual parameter may be any Pointer type.

d) They are both value or variable conformant array specifications (see section 8.5.4) with the same number of parameters and equivalent conformant array schemas. Equivalent means that each of the following propositions hold true:

1) Both schemas have the same number of index type specifications.
2) The ordinal-type-name must represent the same type in both index type specifications.
3) The components of both schemas must have the same type or likewise be equivalent schemas.
4) Both schemas must be packed or both must be unpacked.

Example of a program with procedural parameters

```
PROGRAM calculator (Input, Output);

VAR
   a, b, c: Integer;

PROCEDURE addition
   (    summand_1, summand_2 : Integer;
    VAR total                : Integer);
BEGIN
   total := summand_1 + summand_2;
END {addition};

PROCEDURE division
   (    dividend, divisor : Integer;
    VAR quotient          : Integer);
BEGIN
   quotient := dividend DIV divisor;
END {division};
...
{ additional operations could be declared here }
...

PROCEDURE calculate_and_output
   (left_operand, right_operand: Integer;
    PROCEDURE operation (x, y: Integer; VAR k: Integer));
VAR
   result   : Integer;
BEGIN
   operation (left_operand, right_operand, result);
   writeln (result);
END {calculate_and_output};

BEGIN {main program}
   calculate_and_output (4711, 15, addition);
   calculate_and_output (2048, 32, division);
END {calculator}.
```

Note

When "calculate_and_output" is called, the name of the subprogram to be transfer-red (addition/subtraction) is passed as an actual parameter.
The parameter names of the procedural parameter "operation (x, y, z)" are only required in their parameter list.

Cross-references

Generic pointer type:	6.5.2
Type equivalence:	6.6.1
Procedure heading:	8.1
Function heading:	8.2
INLINE subprograms:	8.3
Directives:	8.6
Required subprograms:	15

Conformant Array Parameters

Conformant array parameters represent the extended requirement of the Standard (level 1). They make it possible to process array parameters of different sizes within subprograms.

The syntax for conformant array parameters is as follows:

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

variable-conformant-array-specification
               = "VAR" identifier-list ":" conformant-array-schema.

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

conformant-array-schema
               = packed_conformant-array-schema
               | unpacked_conformant-array-schema.

packed_conformant-array-schema
               = "PACKED" "ARRAY" "[" index-type-specification "]"
                 "OF" type-name.

unpacked_conformant-array-schema
               = "ARRAY" "[" index-type-specification {";"
                          index-type-specification} "]" "OF"
                          ( type-name | conformant-array-schema ).

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

Conformant array parameters are formal parameters of ARRAY types for which, within the conformant array schema, a canonical index type and a fixed component type are defined. An index type specification defines identifiers which can be used within the subprogram as bound-identifiers (see example below).

When a subprogram with conformant array parameters is called, the index type of the particular actual parameter may be any subrange type of the canonical index type of the corresponding conformant array parameter specification. The value of the bound-identifier is derived from the least and greatest values of the index type of the actual parameter. This definition is referred to as "conformability" (as opposed to assignment-compatibility, which is required when passing variable parameters).

Regarding the distinction between value and variable parameters, see the remarks given in sections 8.5.1 and 8.5.2.

Abbreviated notation for multi-dimensional conformant array schemas

If a conformant array schema closest-contains another conformant array schema, it may be written in an abbreviated form. This short form substitutes a semicolon for the character string "] OF ARRAY [" in the long form. The short form and the long form are equivalent.

Example

```
  ARRAY [u..v : t1] OF ARRAY [j..k : t2] OF t3
```

```
and
```

```
  ARRAY [u..v : t1;           j..k : t2] OF t3
```

are equivalent conformant array schemas.

Actual parameters

When the subprogram is called, an ARRAY-object must be passed for each formal conformant array parameter as the actual parameter. This ARRAY-object must satisfy the following conditions:

a) If the formal parameter is a variable conformant array schema, the actual parameter must be a variable-object (variable access).

b) The actual parameters which are to be assigned to formal parameters from a single conformant array parameter specification must all have the same type.

c) The type of the actual parameters must conform to the conformant array schema (see below).

d) When value parameters are passed, it is not permitted for an actual parameter to be a conformant array parameter.

Conformability of conformant arrays

Let T1 be an ARRAY type with a single index type, and T2 the ordinal type in the index type specification of a conformant array schema. T1 conforms to the conformant array schema when all of the following conditions are met:

a) The index type of T1 is compatible with T2.

b) The least and greatest values of the index type of T1 lie within the value range of type T2.

c) The component type of T1 represents the same type as the type identifier in the conformant array schema, or it is conformable with the conformant array schema in the conformant array schema.

d) Either both T1 and the conformant array schema are unpacked, or they are both packed.

Possible runtime errors:

`Index_Error`	`- In a conformant array parameter, the index type of the actual parameter is not a subrange of the canonical index type of the conformant array schema.`

Example

```
PROGRAM add_vector (Input, Output);

TYPE
   range = -1000..1000;

VAR
   field_1, field_11, field_111: ARRAY [    1.. 100] OF Integer;
   field_2, field_22, field_222: ARRAY [  -10..  10] OF Integer;
   field_3, field_33, field_333: ARRAY [-1000..1000] OF Integer;

PROCEDURE add_fields (VAR v1,v2,v3 : ARRAY [ug..og : range] OF Integer);

VAR
   i : range;

BEGIN
   FOR i := ug TO og DO
   v3[i] := v1[i] + v2[i];
END {add_fields};

BEGIN {main program}
   ... {input the fields}
   add_fields (field 1, field 11, field 111);   {ug is 1, og is 100 }
   add_fields (field 2, field 22, field 222);   {ug is -10, og is 10}
   add_fields (field 3, field 33, field 333);   {ug is -1000, og is 1000}
   ... {output the fields}
END {add_vector}.
```

Both identifiers in the index type specification (in the example, "ug" and "og") are called bound identifiers. The objects for which they stand are neither constants nor variables, but instead represent a separate class of objects which, semantically, are permanently linked to their applied occurrence within a conformant array schema. Values may not be assigned to them, nor may they be used (as constants) in a type definition. They may only be accessed for reading.

Cross-references

Packed types:	6.3
ARRAY type:	6.3.1
Type equivalence:	6.6.1

Directives and Procedure/Function Identifications

A directive stands for the block of the subprogram which may follow further down in the text in the same declaration part, or which is written in a non-Pascal language and does not belong to the Pascal-XT program text.

The only directive prescribed by Standard Pascal is "Forward".
Pascal-XT offers additional directives for accommodating subprograms not written in Pascal. These directives are compiled separately, and then have to be linked into the Pascal-XT program.

If, in a procedure or function declaration, a directive other than Forward is specified, there must not be an associated procedure or function identification.

Directive	Meaning
Forward	The associated subprogram block is specified further down in the program (see below).
C	The subprogram is written in C.
Cobol	The subprogram is written in COBOL.
Fortran	The subprogram is written in FORTRAN.
External	The subprogram is not written in Pascal-XT and is called via a system-specific subprogram interface.
Internal	The subprogram is not written in Pascal-XT. Unlike the directive External, the call is made to the internal call mechanism of Pascal-XT subprograms. As far as the interfaces are concerned, the compiler expects it to react like a subprogram written in Pascal.

Table 8-1 Directives in Pascal-XT

- **Directive Forward**

The directive Forward is required when only the procedure heading or function heading is to be declared. This can be done by using the procedure-identifier or function-identifier before the associated procedure-block is declared. This directive is required to enable mutually recursive subprogram calls (see example).

The identifier of a procedure or function declaration which has been given the directive Forward must occur exactly once in a procedure or function identification belonging to the same declaration part (see section 12.1).

Example

```
PROCEDURE p2 (k : Integer); Forward;

PROCEDURE p1 (j : Integer);
BEGIN
   Writeln (j);
   IF j>0 THEN p2 (j-1);
   . . .
END {p1};

PROCEDURE p2;
BEGIN
   Writeln (k);
   IF k>0 THEN p1 (k-1);
   . . .
END {p2};
```

Since the procedures p1 and p2 call each other, it is necessary (in accordance with 12.1) that p1 be declared before p2 and p2 before p1. By using the directive Forward it becomes possible to make the procedure heading of p2 known before p2 is fully declared. In this way, the outwardly visible part of p2 is already known and can now be used.
The block of p2 does not follow until p1 has been declared, in which case the duplicate specification of the formal parameter list (of procedure p2) may be omitted (it must be omitted in standard Pascal). However, repetition of the formal parameter list, which is permitted in Pascal-XT, is recommended for reasons of clarity, since, among other things, the block follows long after the Forward declaration and one can spare oneself much laborious paging through compiler listings or on the screen. If programming is to be in accordance with standard, the formal parameter list can be enclosed in comment braces.

Note

Between the Forward declaration and the associated identification there may be any number of other declarations (LABEL, CONST, TYPE, VAR, PROCEDURE and FUNCTION declarations). This accords with the extension to the Standard described in chapter 12, whereby declarations may occur in any order.

- **Other directives**

If one of the directives C, Cobol, Fortran, External or Internal is specified in a procedure or function declaration, there must not be an associated procedure or function identification.

For external subprograms of this sort there may be a number of implementation-defined restrictions.

Implementation-defined characteristics

- A Pascal-XT implementation need not support any directives except Forward.
- For subprograms containing one of the directives C, Cobol, Fortran, External or Internal, there may be implementation-defined restrictions on the sort, type and number of parameters used.

Cross-references

Directives:	3.4
Procedure identification:	8.1
Function identification:	8.2
Formal parameter list:	8.5
Declaration part:	11.1
Block:	12.1

Subprogram Calls

Subprogram calls have the following syntax:

```
function-call  = function-name [actual-parameter-list].

procedure-call = procedure-name [actual-parameter-list].

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

actual-parameter = variable-object | expression | type-name
                 | procedure-name  | function-name
                 | package-identifier | expression ":" format-denoter.

format-denoter   = integer-expression [":" integer-expression].
```

A procedure call (procedure statement) causes the block belonging to the called procedure to be executed.

A function call (function designator) returns a value (the function result) which is calculated by executing the function block and represented by the function-name.

If the subprogram has formal parameters, the subprogram call must contain an actual parameter list with the actual parameters to be substituted for the corresponding formal parameters:

a) For each value parameter an expression must be specified. A format denoter is permitted only for the required procedures Write, Writeln and Writestring.

b) For each variable parameter a variable-object must be specified (i.e. a variable access; see chapter 9).

c) For each procedural or functional parameter a procedure-name or function-name must be specified.

d) A type-name may be specified only in the case of required attribute functions (see section 15.9).

e) A package-identifier may only be specified in conjunction with the required procedure Elaborate (see section 15.12).

A subprogram call performs the following actions:

– Evaluates the actual parameters and assigns them to the formal parameters
– Allocates a memory area for the local variables and parameters of the procedure or function
– Executes the statements in the procedure or function block
– Passes the results in the case of functions
– Releases the allocated memory area.

The actions involved in executing a block are described in detail in section 13.3.

Possible runtime errors:

Sections 8.5.1, 8.5.4 and 13.3 describe further errors which may occur when passing parameters and when calling a procedure or function.

`unpredictable effects`	`- The result of a function will be unpredictable after function block execution if the function identifier has not been assigned a value.`

Implementation-dependent characteristic

In a procedure or function call, the sequence in which access, evaluation and actual parameter assignment takes place is implementation-dependent.

Examples of procedure calls

```
swap (aktx, akty);
   { call the procedure swap, defined in section 8.1, }
   { with suitable actual parameters aktx, akty   }

Proc1;

trans (a, x, y);

bisect (fct (x + y), 2.0, 100, y);
```

Examples of function calls

```
s := add (aktx, akty);
     { call the function add, defined in section 8.2, }
     { with two actual parameters aktx and akty }

value := func (x);

bool := f (x, y, z);

s := g (x) + h (y);

Writeln ('Factorial is ', fac(i));
```

Cross-references

Object:	9.6
Variables:	7.1, 9.6
Expression:	9
Block:	12.1
Write, Writeln:	15.1
Writestring:	15.3
Attribute functions:	15.9
Elaborate:	15.12

Expressions

General Remarks

An expression is an arithmetic formula which returns a value when it is computed, unless an exception occurs during the computation or the expression contains a function call and the associated function block is exited by means of a GOTO statement. The value of the expression depends on the values of its operands (variables, constants, function calls) and on the operators involved.

An expression has the following syntax:

```
expression     = simple_expression [ relational_operator simple_expression ].

relational_operator
               = "=" | "<>" | "<" | ">" | "<=" | ">=" | "IN" .

simple_expression
               = [ sign ] term { adding_operator term } .

sign           = "+" | "-" .

adding_operator
               = "+" | "-" | "OR" | "OR" "ELSE" .

term           = factor { multiplying_operator  factor } .

multiplying_operator
               = "*" | "/" | "DIV" | "MOD" | "AND" | "AND" "THEN" .

factor         = primitive [ "**" primitive ] | "NOT" factor .

primitive      = unsigned_constant
                | bound-identifier
                | "(" expression ")"
                | set=constructor
                | qualified_set_constructor
                | object .

unsigned_constant
                = constant_name
                | unsigned_integer_number
                | unsigned_real_number
                | character_string
                | "NIL" .
```

In Pascal-XT, the exponent operator ** has been introduced as an extension to the Standard. It has the same priority as the NOT operator. As a result, Pascal-XT has been given the concept of a primitive. Also as an extension to the Standard, Pascal-XT recognizes the shortcut operators AND THEN and OR ELSE.
A primitive may also be a "qualified set constructor" (see section 9.4) or an "object" (see section 9.6), which likewise represent extensions to Standard Pascal.

Bound identifiers play a role in conjunction with conformant array parameters (see section 8.5.4).

Primitives, factors, terms and simple expressions are all referred to below as operands.

Expressions are computed in according with the rules of algebra. Since expressions consist of a sequence of operands and operators, the sequence in which the individual operations are executed is of crucial importance, and is defined as follows:

- The evaluation of an expression consists of a series of operations with operands and operators, where operators with the highest precedence are executed first (see section 9.3).
- A sequence of operators with identical precedence is processed from left to right (left associativity).
- Expressions in brackets are evaluated first.
- These rules likewise apply to expressions within brackets.

The evaluation sequence of operands in dyadic operators is not defined (see implementation-dependent characteristics in section 9.3).

The value of an expression is not defined if the expression contains a variable access (as an object in a primitive) which has not yet been assigned a value (undefined variable, see section 7.3).

If the type of the object of a primitive is a subrange type, the type of the primitive is the host type of the subrange type. If the type of the object of a primitive is a SET type SET OF t, the type of the primitive is SET OF w, where w is either the host type of type t or, if t is a subrange of Integer, an implementation-defined subrange of Integer (see section 6.3.4).
In all other cases, the type of the primitive is identical to the type of the object.

Possible runtime errors:

`unpredictable effects`	`- A variable access used as an object in an expression has an undefined value at the time the (sub)expression is evaluated (see also section 7.3).`

Examples of primitives

```
-  Set constructor                [1,5,x..y,z+23]
-  Qualified set constructor      set_of_t_type ([1, 2])
-  Function call                  Sin (x+y)
```

Examples of factors

```
-  Negated factor                 NOT (a = b)
-  Exponential                    x ** 3
```

Examples of terms

```
x * y
i / (j - i)
(x <= y) AND (y < z)
(p <> NIL) AND THEN (p↑.i > 0)
```

Examples of simple expressions:

```
x + y
-x
i * j + 1
b OR (x = y)
(Length (s) = 0) OR ELSE (s[1] = '.')
```

Examples of expressions:

```
x = 1.5
p <= q
(a + b) > (a * b)
next_char IN letters
```

Cross-references

Unsigned Integer number:	3.5
Unsigned Real number:	3.5
Character string:	6.3.2
Constants:	5
Variables:	7
Undefined value:	7.3
Bound identifier:	8.5.4
Conformant array parameter:	8.5.4
Operators:	9.3
Set constructors:	9.4
Object:	9.6.1
GOTO statement:	10.1.4
Block:	12.1
Exception:	14

Static Expressions

A static expression is an expression whose value can already be evaluated at compile time. Consequently, a static expression must not contain a variable access or a function call.
The following required subprograms may occur in a static expression if their actual parameters are in turn static expressions:

```
Abs       Long      Round     Sqr       Trunc
Short_Round   Long_Round   Short_Trunc   Long_Trunc
Chr       Ord       Odd       Pred      Succ
Card      Setmax    Setmin
Concat    Length    Substring
Alignof   Bitsizeof First    Last       Maxlength  Sizeof
Convert
```

The Convert function is governed by further restrictions which are described in section 15.10.

In Pascal-XT (as an extension to the Standard), static expressions may be used wherever the standard only permits constants.

The sections below describe when each particular expression is static.

Examples of static expressions

```
  2 * 3.14159
  Maxint - 1
  Length ('String-constant')
  2 ** 8 -1
  Concat (#'14', 'highlighted', #'15')
  Card (['a'..'z']  { does not always yield 26 }
                    keine Akkolade
  today.year        { see static RECORD aggregate in section 9.5.2 }
  null_vector[1]    { see static ARRAY aggregate in section 9.5.1 }
```

Cross-references

Constants:	5
Subprograms:	15

Operators

The syntax governing expressions, simple expressions, terms, factors and primitives prescribes rules of precedence for the operators.

– The operator NOT and the exponent operator ** take precedence over all other operators.

– Then come the multiplying operators.

– Then come the adding operators and signs.

– The lowest precedence is given to relational operators.

Sequences of two or more operators of the same precedence are evaluated from left to right (left associativity). These rules of precedence may be circumvented by using parentheses.

Implementation-dependent characteristics

Except for the two shortcut operators OR ELSE and AND THEN, the sequence for calculating the operands in a dyadic operator is implementation-dependent. The operands may be evaluated in the order in which they were written, in the reverse order, simultaneously, or perhaps not at all.

Examples:

```
a + b * c          is equivalent to a + (b * c), as opposed to
(a + b) * c

NOT b1 OR b2       means (NOT b1) OR b2, as opposed to
NOT (b1 OR b2)

a = b AND c = d    would mean (a = (b AND c)) = d, and should
                   therefore be parenthesized:
(a = b) AND (c = d)
```

Arithmetic Operators

Tables 9-1 and 9-2 provide an overview of the permissible operand types for monadic and dyadic operations. Tables 9-3 to 9-6 give a detailed description of the result types for dyadic operators.

Operator	Operation	Type of operand	Type of result
+	Identity	Integer	Integer
		Real	Real
		Short_Integer	Integer
		Long_Integer	Long_Integer
		Short_Real	Short_Real
		Long_Real	Long_Real
-	Sign inversion	Integer	Integer
		Real	Real
		Short_Integer	Integer
		Long_Integer	Long_Integer
		Short_Real	Short_Real
		Long_Real	Long_Real

Table 9-1 Monadic arithmetic operators (signs)

Operator	Operation	Type of operand	Type of result
+	Addition	Integer or Real	see Table 9-3
-	Subtraction	Integer or Real	see Table 9-3
*	Multiplication	Integer or Real	see Table 9-3
/	Division	Integer or Real	see Table 9-4
**	Exponentiation	left: Integer or Real right: Integer	see Table 9-5
DIV	Division with truncation	Integer	see Table 9-6
MOD	Modulo	Integer	see Table 9-6

Table 9-2 Dyadic arithmetic operators

Note

The symbols +, -, * and / are also used as set operators (see section 9.3.3).

- **Integer operations**

For operations on values of type Integer, the following properties apply:

The monadic arithmetic operations "+" and "-", as well as the dyadic operations "+", "-", "*", "**", "DIV" and "MOD", are executed in accordance with the rules of mathematics when the operands have Integer values and the result lies in the value range of the result type (as per tables 9-1 and 9-2).

A Numeric_Error (see below) occurs when the result value of an arithmetic integer operation lies outside the value range, i.e. when an overflow occurs.

Notes

– In Pascal-XT, an error can also occur with the monadic operation "-". The value of the expression "-Long_Minint", for example, does not lie within the value range Long_Integer.

– If subtotals in expressions lie outside the relevant value range, but they can nevertheless still be used to produce mathematically correct calculations, it is implementation-dependent whether to proceed with the calculation or issue a Numeric_Error.

– A dyadic operation with operands of type Short_Integer returns a value of type Integer, which may be implementation-defined as Short_Integer or Long_Integer. This should be borne in mind when porting programs as the programs may otherwise behave differently.

– To avoid overflows with operands of type Short_Integer, at least one of the operands must be converted with the required function Long (see section 15.5) into a value of the type Long_Integer. For example:

```
x := y * Long (z)
```

– An expression which consists entirely of Integer type operands, and whose value is to be assigned to a Real variable, is calculated in integer arithmetic and is not converted into a Real value until the assignment takes place.

- **Real operations**

Real operations are monadic and dyadic arithmetic operations with at least one Real-type operand, or the operator "/" with Integer-type operands.

Real-type values form subsets of the real numbers. The results of arithmetic real operators are therefore, in general, only approximations of the corresponding mathematical results (see section 6.2.2). Depending on the hardware used, real operations can also be calculated internally at a higher level of precision. In this case a loss of precision may occur when the values are stored. Relational operators may, for example, yield different values depending on whether variables are compared with each other or with the results of an expression. In addition, the rounding behavior of the machine influences the result. In particular, comparisons for equality should be avoided.

A Numeric_Error occurs when the approximate value of the result of a real operation lies outside the value range of the result type (as per tables 9-1 and 9-2). Depending on the implementation used, subtotals may also be computed with greater precision and/or with a larger value range.

Operations with Real constants

The type of a Real constant is a so-called universal Real type, whose value range is identical to that of Long_Real. If, in a dyadic operation, the type of an operand is of the universal Real type, the operation is executed with the level of precision defined by the type of the other operand. If both operands are of the universal Real type, the calculation will use the level of precision appropriate to the type Long_Real.

Mixtures of Integer and Real operands

If, in a dyadic operator, one of the operands is of an Integer type and the other is of a Real type, the first thing that happens is an implicit type conversion. The value of the Integer operand is converted into a real number having the universal Real type.

- **The arithmetic operations "+", "-" and "*"**

Type of operands		Type of result
x	y	x (+|-|*) y
Integer	Integer Real	Integer Real
Real	Integer Real	Real Real
Short Integer	Short_Integer Long_Integer Short_Real Long_Real	Integer Long_Integer Short_Real Long_Real
Long_Integer	Short_Integer Long_Integer Short_Real Long_Real	Long_Integer Long_Integer Short_Real Long_Real
Short_Real	Short_Real Long_Real Integer type	Short_Real Long_Real Short_Real
Long_Real	Real_type Integer type	Long_Real Long_Real
universal Real type	universal Real type Long_Real Short_Real Integer type	universal Real type Long_Real Short_Real universal Real type

Table 9-3 Result type for operations "+", "", "*"

- **Division operator "/"**

With division, the result is always of a Real type.

Type of operands		Type of result
x	y	x / y
Integer	Integer Real	Real Real
Real	Integer Real	Real Real
Short_Real	Short_Real Long_Real Integer type	Short_Real Long_Real Short_Real
Long_Real	Real type Integer type	Long_Real Long_Real
Integer type	Short_Real Long_Real Integer type	Short_Real Long_Real Long Real
universal Real type	universal Real type Long_Real Short_Real Integer type	universal Real type Long_Real Short_Real universal Real type

Table 9-4 Result type for operation "/"

- **The exponent operation "**"**

Type of operands		Type of result
x	n	x ** n
Integer Real	Integer Integer	Integer Real
Short_Integer Long_Integer	Integer_type Integer type	Integer Long_Integer
Short_Real Long_Real	Integer type Integer type	Short_Real Long_Real
universal Real type	Integer type	universal Real type

Table 9-5 Result type for exponentiation

- **The "DIV" and "MOD" operators**

Type of operands		Type of result
x	y	x (DIV|MOD) y
Integer	Integer	Integer
Short_Integer	Short_Integer Long_Integer	Integer Long_Integer
Long_Integer	Integer type	Long_Integer

Table 9-6 Result type for integer operations "DIV" and "MOD"

The following applies with regard to the DIV operator:

```
Abs(x) - Abs(y) < Abs (x DIV y) * y <= Abs(x)
```

The result is zero if Abs(x) < Abs(y). The result is positive if x and y have the same sign, and negative if their signs are different.

Examples

```
   5 DIV 3   yields  1
  -5 DIV 3   yields -1

   5 MOD 3   yields  2
   4 MOD 3   yields  1
   3 MOD 3   yields  0
   2 MOD 3   yields  2
   1 MOD 3   yields  1
   0 MOD 3   yields  0
(-1) MOD 3   yields  2
(-2) MOD 3   yields  1   (etc.)
(-3) MOD 3   yields  0   (The remainder is determined to the next
                          smallest multiple of the divisor.)
```

In various instances, execution of an arithmetic operation will cause the occurrence of a Numeric_Error:

Possible runtime errors:

`Numeric_Error`	`- y = 0 in an expression of the form x/y.`
	`- j = 0 in an expression of the form i DIV j.`
	`- j <= 0 in an expression of the form i MOD j.`
	`- The result of an arithmetic operation does not lie in the value range of the result type.`
	`- In an expression of the form x**n,`
	`  -  x is of an Integer type and n < 0, or`
	`  -  x = 0 or 0.0 and n <= 0.`

Cross-references

Errors:	2.3, A.5
Constants:	5
Integer types:	6.2.1
Real types:	6.2.2

Boolean Operators

Operands and results of Boolean operators have the type Boolean.

Operator	Operation	Type of operands	Type of result
OR	logical disjunction	Boolean	Boolean
AND	logical conjunction	Boolean	Boolean
NOT	logical negation	Boolean	Boolean
OR ELSE	logical disjunction	Boolean	Boolean
AND THEN	logical conjunction	Boolean	Boolean

Table 9-7 Boolean operators

a	b	a OR b
False	False	False
False	True	True
True	False	True
True	True	True

Disjunction

a	b	a AND b
False	False	False
False	True	False
True	False	False
True	True	True

Conjunction

a	NOT a
False	True
True	False

Negation

Table 9-8 Truth values for disjunction, conjunction and negation

As explained in section 9.3, the sequence in which the operands are evaluated for OR and AND is implementation-dependent.
For the shortcut operators AND THEN and OR ELSE, on the other hand, the left-hand operand is always evaluated first, while evaluation of the right-hand operand depends on the result provided by the left-hand operand (shortcut).

a OR ELSE b	If the value of a is True, b is not evaluated and the result is True.
a AND THEN b	If the value of a is False, b is not evaluated and the result is False.

One area of application for shortcut operators is, for instance, the processing of lists when other criteria are given in addition to the terminate criterion end-of-list. When end-of-list is reached, the subsequent operands must under no circumstances be evaluated if they involve pointer dereferencing (see example 1).

Example 1

Processing of the list "list" is to terminate when end-of-list is reached or an element is located whose component "number" has the value 1.

```
WHILE (list <> NIL) AND THEN (list↑.number <> 1) DO
   list := list↑.next;
```

Example 2

An action should take place when the end of an array is reached or a negative element is encountered:

```
IF (index > max_index) OR ELSE (a[index] < 0) THEN ...
```

Cross-references

Boolean type:	6.2.4
Evaluation:	9.3

Set Operators

The table below summarizes the rules governing the types of the operands and results of set operations.

Operator	Operation	Type of operands	Type of result
+	Set union	SET type	Type of operands
-	Set difference		
*	Set intersection		
/	Symmetrical set difference		

Table 9-9 Set operations

In a set operation, the types of the two operands must be compatible with each other, i.e. they must have compatible base types and they must be either both packed or both unpacked. If one of the operands is an unqualified set constructor, its type is pak-ked or unpacked depending on the type of the other operand (see section 9.4).

In addition Pascal-XT recognizes the symmetrical set difference "/" (equivalent to exclusive OR). This operator is defined as follows:

A / B = (A - B) + (B - A)

Notes

- Set operations cannot be executed in Pascal-XT if the implementation-defined maxi-mum number of values of the base type of the set is exceeded. Thus, for example, the union and the symmetrical difference of two sets can result in a set which is too large.
- The symbols "+", "-", "*" and "/" are also used as arithmetic operators.

Examples

```
CONST
   digits     = ['0' .. '9'];
   hexletters = ['a','b','c','d','e','f'];
   hexdigits  = digits + hexletters;

VAR
   set1 : SET OF 0 .. 200;
   set2 : SET OF 100 .. 300;
   set3 : SET OF 0 .. 1000;

BEGIN
   set1 := [0, 10..20, 100, 200];
   set2 := [99, 101, 300];
   set3 := set1 + set2;        { = [0, 10..20, 99..101, 200, 300] }
   set3 := set1 * set2;        { = [], the empty set }
END
```

Cross-references

Compatibility:	6.6.2
SET type:	6.3.4
Base type:	6.3.4
Packed/Unpacked:	6.1
Set constructor:	9.4

Relational Operators

The table below summarizes the rules governing the operands and results of relational operations.

Operator	Type of operands	Type of result
`= <>`	Ordinal, Real, Pointer, generalized string or SET type	Boolean
`<   >`	Ordinal, Real, generalized string or SET type	Boolean
`<=  >=`	Ordinal, Real, generalized string or SET type	Boolean
`IN`	Left operand: an ordinal type Right operand: a SET type	Boolean

Table 9-10 Relational operations

Meaning of the relational operators:

```
u =  v   u is equal to v
u <> v   u is not equal to v
u <  v   for ordinal types and Real types: u is less than v
u <  v   for SET types: u is a genuine subset of v (i.e. not identical)
u >  v   for ordinal types and Real types: u is greater than v
u >  v   for SET types: v is a genuine subset of u (not identical)
u <= v   for ordinal types and Real types: u is less than or equal to v
u >= v   for ordinal types and Real types: u is greater than or equal to v
u <= v   for SET types: u is a subset of v
u >= v   for SET type: v is a subset of u
a IN v   a is a member of the set v
```

- **IN operator**

The operator IN returns the value True if the value of the left-hand operand is a member of the value of the right-hand operand; otherwise, it returns the value False.

The ordinal type of the left-hand operand must be compatible with the base type of the SET type of the right-hand operand.

- **Comparison of simple values**

Either the types of the operands must be compatible, or one of the operands must be of a Real type and the other of an Integer type.

Since Real-type numbers only represent approximations of real values, comparison between two Real-type numbers, or between a Real-type number and an Integer-type number, may not yield the expected result. In particular, interrogation for identity ("=") or non-identity ("<>") should be avoided in the case of Real-type operands.

- **Comparison of pointers**

The Pointer types must be compatible, i.e. the two operands must have the same type (see section 6.6.1), or one of the two operands must have the generic pointer type (see section 6.5.2). Only the relational operators "=" and "<>" may be applied to Pointer types.

In other words, only those pointers may be compared that point to identified variables of the same type. Comparison with "=" and "<>" can be used to determined whether two pointers point to the same identified variable or whether a Pointer variable has the value NIL.

- **Comparison of sets**

The SET types of the two operands must be compatible. In other words, they must have compatible base types and they must either both be packed or both unpacked. If one of the two operands is an unqualified set constructor, its type is considered unpakked unless the SET type of the other operand is packed.

As an extension to standard, Pascal-XT also permits the relational operators "<" and ">" (genuine subset or genuine superset).

- **Comparison of character strings**

When characters strings are compared, their types must be compatible. Table 9-11 lists the possible combinations of operands, where PACKED ARRAY is short for fixed string types (see section 6.3.2.1) and String [n] or String [m] is short for the variable string types (see section 6.3.2.2).

	PACKED ARRAY [1..n] OF Char	String [m]	Char
PACKED ARRAY [1..k] OF Char	if k = n also k <> n	yes	no
String [n]	yes	yes	yes
Char	no	yes	yes

Table 9-11 Compatibility of operand types for comparison of character strings

Comparison of two character strings takes place in accordance with the lexicographical comparison described below. This lexicographical comparison defines a total ordering of the set of all character strings. Comparison occurs character-by-character from left to right, whereby the comparison of corresponding characters is governed by the manner in which the implementation-defined characters of type Char are ordered.
If the actual lengths of s1 and s2 are not identical, comparison uses the length of the shorter operand. If this comparison yields identity, the string with the shorter actual length is considered the smaller of the two.

Let s1 and s2 be two character strings. Let the actual length of s1 be n1 and that of s2 be n2, and let n be the smaller value of n1 and n2. The following now holds true:

```
s1 = s2 only if n1 = n2 and for all i in [1..n]:
                         s1[i] = s2[i].

s1 < s2 only if there is a p in [1..n]
                and for all i in [1..p-1]:
                         s1[i] = s2[i] and s1[p] < s2[p],
                or if for all i in [1..n]:
                         s1[i] = s2[i] and n1 < n2.

s1 >  s2 is equivalent to s2 < s1
s1 <= s2 is equivalent to (s1 < s2) OR (s1 = s2)
s1 >= s2 is equivalent to (s1 > s2) OR (s1 = s2)
s1 <> s2 is equivalent to NOT (s2 = s1)
```

Example

Table 9-12 illustrates the evaluation of the expression "a relational-operator b", where a and b are character strings:

a	b	<	=	<=	>	>=	<>
'ABC'	'ABCD'	True	False	True	False	False	True
'ABCD'	'ABC'	False	False	False	True	True	True
'ABCD'	''	False	False	False	True	True	True
'ABC'	'BBB'	True	False	True	False	False	True
'A'	'A'	False	True	True	False	True	False
'B'	'AAAA'	False	False	False	True	True	True
'B'	'A'	False	False	False	True	True	True
'A'	'A '	True	False	True	False	False	True

Table 9-12 Comparison of character strings

Cross-references

Packed/Unpacked:	6.1
Ordinal types:	6.2
Real types:	6.2.2
Generalized string types:	6.3.2
Set types:	6.3.4
Pointer types:	6.4
Type equivalence:	6.6.1
Type compatibility:	6.6.2

Set Constructors

The syntax of set constructors is as follows:

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

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

qualified-set-constructors
                = type-name "(" set-constructor ")".
```

A set constructor stands for a value of type SET. The set constructor "[ ]", without member designators, is referred to as the empty set, and is a value of each and every SET type.

A set constructor represents a value containing zero, one or more members. Each member is specified by means of at least one of the member designators of the set constructor. A member designator consisting of a single expression stands for the value which the expression possesses. A member designator in the form `a .. b` describes an interval, i.e. all values x where a ≤ x ≤ b. If `a > b`, then the member designator a..b does not describe a member.

A set constructor containing one or more member designators stands for a value of the type SET OF w, where w is the common (host) type of all expressions occurring in the member designators. This host type must be an ordinal type.
The size of the host type may be given an implementation-defined limit. In this case, a maximum permissible subrange of the host type, whose least ordinal value is 0 and whose greatest ordinal value is implementationdefined, is assumed as the base type of the SET type.

The type of an (unqualified) set constructor is considered to be unpacked unless a pakked SET type is required by the context. This makes it possible to assign set constructors to packed and unpacked SET variables and to link them by means of set and relational operators.
However, when a set constructor occurs as a constant in a constant definition, there is no context requiring a packed SET type. For this reason, the constant defined in this manner has an unpacked SET type (see also section 5.1).

- **Qualified set constructors**

Pascal-XT also recognizes the qualified set constructor, whose type is determined explicitly by specifying a type-name rather than implicitly from the host type of all member designators. The values for which the member designators of a qualified set constructor stand must be assignment-compatible with the base type of the specified SET type.

Implementation-defined characteristic

The greatest ordinal value of the base type of a SET type of an unqualified set constructor is implementation-defined.

Implementation-dependent characteristic

The sequence in which the member designators and the expressions contained therein is evaluated in a set constructor is implementation-dependent.

Possible runtime errors

`Set_Error`	`- In a set constructor, the value of a member designator does not lie in the value range of the base type of the set constructor.`

Notes

- In Pascal-XT, the maximum number of values of the base type of a set may be limited (see section 6.3.4).
- Qualified set constructors are required especially when, for a SET OF t, the base type t is a subrange of type Integer that is not fully contained in the subrange 0 .. n, where n is the implementation-defined greatest ordinal value of the base value of a SET type.
- The sequence in which the member designators are specified in a set constructor is arbitrary; no sequence is prescribed.

Examples

```
[ ]                           { the empty set }

[40, 49, 11..33])             { sequence of values is arbitrary }

['A', 'E', 'I', 'O', 'U']     { contains exactly these characters }

[True, False]                 { all values of data type Boolean }

[red, yellow]                 { only the values "red" and "yellow" }

[1..5, x..y, a + b, 17, 31]   { variables and expressions in
                                member designators }

[f(x)..f(y)]                  { function calls in member designators }

[-10]                         { qualified set constructor required as
                                -10 does not lie in interval 0 .. n }

st1 ([-10, 10])               { qualified set constructor, e.g.
                                st1 = SET OF -10..10 }

st2 ([8000..9000])            { qualified set constructor, e.g.
                                st2 = SET OF 8000..9000 }
```

Cross-references

Ordinal type:	6.2.2
Host type:	6.2.6
Subrange:	6.2.6
Base type:	6.3.4
SET type:	6.3.4
Assignment-compatible:	6.6.3
Expression:	9

Aggregates

Aggregates are used to form RECORD- and ARRAY-type values from the values of their components (see sections 9.5.1 and 9.5.2).

```
aggregate = ARRAY-aggregate | RECORD-aggregate .
```

An aggregate may be located at the right-hand side of a constant definition as a static expression. In this way, it also becomes possible to define RECORD and ARRAY constants.

An aggregate whose aggregate members are all static expressions is called a static aggregate.

Implementation-dependent characteristic

The sequence in which the expressions in an aggregate are evaluated, and in which its type is assigned to the components, is implementation-dependent.

The value of each aggregate member must be assignment-compatible with the type of the corresponding RECORD or ARRAY components.

Possible runtime errors:

`Numeric_Error`	`- In an aggregate, the value of an aggregate member of the type Long_Real does not lie in the value range of the Short_Real type of the associated aggregate component.`
`Range_Error`	`- In an aggregate, the value of an aggregate member of an ordinal type does not lie in the value range of the ordinal type of the associated aggregate component.`
`Set_Error`	`- In an aggregate, the value of an aggregate member of a SET type does not lie in the value range of the SET type of the associated aggregate component.`
`String_Error`	`- In an aggregate, the actual length of a character string of an aggregate member is greater than the maximum length of the variable string type of the associated aggregate component.`
	`- In an aggregate, the actual length of a character string of an aggregate member (of a variable string type) is not equal to the length of the associated aggregate component (of a fixed string type).`
`unpredictable effects`	`- In an aggregate, the type of a Pointer value is of the generic pointer type and the Pointer value of the expression points to an identified variable whose type differs from the domain type of the type of the corresponding aggregate component.`

ARRAY Aggregates

ARRAY aggregates have the following syntax:

```
ARRAY-aggregate = ARRAY-type-name "(" ARRAY-aggregate-member
                               { "," ARRAY-aggregate-member } ")".

ARRAY-aggregate-member
                = expression [":" repeat-factor].

repeat-factor   = integer-constant.
```

The values of the ARRAY aggregate members must be assignment-compatible with the component type of the ARRAY type. The values of the index type are assigned the values of the ARRAY aggregate members in ascending sequence in the order they appear in the text. The value of the ARRAY aggregate then consists of the assigned component values.

Any repeat factor in an ARRAY aggregate member must be a static expression of an Integer type whose value is greater than 0. An ARRAY aggregate member containing a specified repeat factor of the value n stands for an n-fold repetition of the same component value. In the ARRAY aggregate there must be exactly as many ARRAY aggregate members (including any repeat factors) as there are values in the index type belonging to the ARRAY type of the aggregate.

If the ARRAY aggregate member has an ARRAY or RECORD type, it must be specified again as an aggregate (see "matrix" in example).

Example

```
TYPE
   vector = ARRAY [1..5] OF Integer;
   matrix = ARRAY [1..5] OF vector;
CONST
   null_vector = vector ( 0: 5 );
   null_matrix = matrix (null_vector : 5);
VAR
   m : matrix;
BEGIN
   m := matrix (null_vector : 5);        { = null_matrix }
   m := matrix (vector (1, 0:4),         { 1 0 0 0 0 }
                vector (0, 1, 0:3),      { 0 1 0 0 0 }
                vector (0:2, 1, 0:2),    { 0 0 1 0 0 }
                vector (0:3, 1, 0) : 2); { 0 0 0 1 0 }
                                         { 0 0 0 1 0 }
END
```

Cross-references

ARRAY type:	6.3.1
Index type:	6.3.1
Component type:	6.3.1
Constant:	5
Assignment-compatibility:	6.6.3
Expression:	9
Static expression:	9.2

RECORD Aggregates

RECORD aggregates have the following syntax:

```
RECORD-aggregate = RECORD-type-name "(" expression {"," expression} ")".
```

The type of a RECORD aggregate must not be an empty RECORD type. At first, the first RECORD aggregate members are assigned to the fields of the fixed part of the field list of the RECORD in the order they appear in the text.

If this field list has a variant part, the variant part must have a tag field, and the next aggregate member must be a static expression. The value of this static expression is then assigned to the tag field, and determines the variant of the value of the RECORD aggregate.

The remaining RECORD aggregate members are then assigned in the same way to the fields of the field list belonging to this variant. The number of RECORD aggregate elements must exactly coincide with the total number of fields in the variants selected in the aforementioned manner and the fields in their field lists.

The value of each RECORD aggregate member must be assignment-compatible with the type of the assigned field or tag field. The value of the RECORD aggregate then consists of the component values assigned in this manner.

If an ARRAY aggregate member has an ARRAY or RECORD type, it must be specified again as an aggregate.

Example

```
TYPE
   rectyp = RECORD
       x : Integer;
       CASE b : Boolean OF
          True  : (u : Integer);
          False : (v,w : Char);
    END;

CONST
   const1  = rectyp (3, True, 5);
   const2  = rectyp (3, False, ’A’, ’B’);

TYPE
   date  = RECORD
     day:   1..31;
     month: 1..12;
     year:  1..2000;
   END;
```

```
CONST
   today  = date (1, 1, 1986);

FUNCTION input_date: date;
VAR
   d, m, y: Integer;
BEGIN
   Writeln ('Please enter day, month and year.');
   Readln; Read (d, m, y);
   input_date := date (d, m, y);
END;
```

Cross-references

RECORD type:	6.3.3
Fixed part, field list:	6.3.3
Tag field, variant part:	6.3.3.1
Assignment-compatibility:	6.6.3
Expression:	9

Data Objects

General Remarks

Besides components of variables, Pascal-XT also includes components of structured constants and structured values which are determined by means of aggregates and function results. For this reason, the data object concept has been introduced as an extension to Standard Pascal with the following general form:

```
object  = constant-name            | variable-name
        | aggregate                | function-call
        | indexed_object           | selected-object
        | dereferenced_object      | buffer-variable
```

Objects are constants, variables, aggregates, results of function calls, or buffer variables. If the types of these objects are structured, then components of these objects can also be accessed. Thus, an indexed object is a component of an object of an ARRAY type or variable string type which may be index-accessed. A selected object is a component of an object of a RECORD type on which access may be performed by selection (field designator). A dereferenced object accesses a dynamically created (identified) variable (see section 9.6.4), which can be accessed by means of pointer dereferencing. A buffer variable is declared implicitly for each FILE variable.

At the following positions objects must be variable accesses (variable objects):

– as an actual parameter if the corresponding formal parameter is a variable parameter;

– as an actual parameter at particular positions in the calls for the required procedures New, Mark, Pack, Unpack, Read, Readln, Readstring, Writestring, Delete and Insert;

– on the left-hand side of an assignment;

– as a RECORD variable in a WITH statement.

The following rules apply:

- Objects which are identified by a constant-name are not variable accesses.
- Objects which are identified by a variable-name are variable accesses.
- Aggregates are not variable accesses.
- Functions calls are not variable accesses.
- The sections below describe the conditions under which indexed and selected objects are variable accesses.

In Standard Pascal the following restrictions apply:

- Only constants of a simple type exist.
- Only a simple type is permitted as the result type of functions.
- Functions with a Pointer type as result type cannot be dereferenced.
- Only variables can have a structured type.
- Only variable-objects (variable accesses) may be indexed, selected or dereferenced.
- In dynamic (identified) objects, the Pointer-object must always be a Pointer-variable.

Cross-references

Variable access:	7
Variable name:	7.1
Identified variable:	7.2
Variable parameter:	8.5.2
Function call:	8.7
Static expression:	9.2
Aggregate:	9.5
Assignment:	10.1.2
WITH statement:	10.5
Required function:	15

Indexed Objects

A component of an ARRAY object is selected by means of indexing. Indexed access is made in accordance with the following syntax:

```
indexed-object   = ARRAY-object
                   "[" index-expression {"," index-expression} "]"
                   | variable-string-object "[" index-expression "]".

index-expression = ordinal-expression.
```

An indexed object is a component of an object of an ARRAY type or a variable string type. The type of the indexed object is the component type of the ARRAY-object, or the type Char in the case of a String-object. If the object is a variable access, the indexed object is likewise a variable access.

In Standard Pascal, only variable accesss can be indexed.

If the type of the object is an ARRAY type, the value of the index expression must be assignment-compatible with the corresponding index type of the ARRAY type. In other words, the value must lie in the value range of the index type.

If the type of the object is a variable string type, the value of the index expression must have an Integer type. The value of the Integer expression must not be less than 1 or greater than the actual length of the String-object.

An indexed object is a static object if and only if the ARRAY-object or the String-object is a static expression and all of the index expressions are static objects.

Abbreviated notation for multi-dimensional ARRAY-objects

The sequence "][" in the long form is replaced in the short form by a comma. The short form and the long form are equivalent.

Example

Given the following declarations:

```
TYPE
   t1  = Integer;
   t2  = ARRAY ['A'..'Z'] OF t1;
   t3  = ARRAY ['0'..'9'] OF t2;
VAR
   v1  : t1;
   v3  : t3;
```

The two assignments shown below are then equivalent:

```
   v1 := v3['0']['A'] ;
   v1 := v3['0', 'A'] ;
```

Implementation-dependent characteristic

The sequence in which index expressions in an indexed object are evaluated is implementation-dependent.

Possible runtime errors:

Index_Error	- In an indexed ARRAY-object, the value of the index expression does not lie in the value range of the index type of the type of the ARRAY-object. - In an indexed *String*-object, the value of the index expression is less than 1 or greater than the actual length of the *String*-object.
unpredictable effects	- In an indexed *ARRAY*- or *String*-object, the *ARRAY*- or *String*-object was generated in short form by calling New(p,e) or New (p, c1, .., cn, e) and the value of the index expression in the indexed object is greater than e. - In an indexed *String*-object, the value of the *String*-object is undefined (regardless of whether the indexed object occurs in an expression, or is used e.g. as a variable access on the left-hand side of an assignment). - The length of a string variable is changed although there still exists a reference to a component of this String variable.

Notes

– Only in exceptional cases should String variables be used in indexed form. Accesses beyond the actual length (but within the declared maximum length) are illegal, and will raise an Index_Error if the Check option is activated. For this reason, the required subprograms for character string processing (see section 15.3) should be used wherever possible.

– Note in particular that the value of a String-variable (and hence its actual length) must be defined, i.e. there must have been an assignment to the (whole) String-variable before its individual components can be index-accessed (for reading or for writing).
Assigning characters to individual components of a String-variable does not change the length of the String-variable.

Examples

```
TYPE
   byte     = 0 .. 255;
   hex_pair = PACKED ARRAY [1..2] OF Char;


FUNCTION hex (i: byte) : hex_pair;
CONST
   hex_base = 16;
TYPE
   tab      = PACKED ARRAY [0 .. hex_base - 1] OF Char;
CONST
   hex_tab  = tab ('0','1','2','3','4','5','6','7',
                   '8','9','A','B','C','D','E','F');

BEGIN
   hex := hex_pair (hex_tab [i DIV hex_base],  ———————————————————————  (1)
                    hex_tab [i MOD hex_base]);
END;


VAR
   i   : byte;
   pair: hex_pair;
   ch,
   ch2 : Char;
   ptr : ↑hex_pair;


BEGIN
   Read (i);
   pair     := hex (i);
   ch       := pair[1];  ————————————————————————————————————————————  (2)
   ch2      := hex (i) [1];  ————————————————————————————————————————  (3)
   New (ptr);
   ptr↑ [1] := ch;  ——————————————————————————————————————————————————  (4)
END
```

(1) The constant hex_tab is of an ARRAY type and may therefore be indexed.

(2) The variable pair is of an ARRAY type and may therefore be indexed.

(3) The result of the function call hex (i) is of an ARRAY type and may therefore be indexed.

(4) The identified variable ptr↑ is of an ARRAY type and may therefore be indexed.

Since even the aggregate is of an ARRAY type and may therefore be indexed, the declaration of the constant hex_tab might have been omitted and replaced throughout by the aggregate. For example:

```
    hex_tab [i DIV hex_base]
```

could have been replaced by the (highly complicated) object

```
    tab ('0','1','2','3','4','5','6','7',
           '8','9','A','B','C','D','E','F') [i DIV hex_base]
```

Cross-references

Component type:	6.3
ARRAY type:	6.3.1
Index type:	6.3.1
Variable string type:	6.3.2.2
Assignment-compatibility:	6.6.3
Expression:	9
Static expression:	9.2
Identified variable:	9.6.4
Assignment:	10.1.2
New:	15.2

Selected Objects

A component of a RECORD-object is singled out by means of selection (field designator). Selected access is performed in accordance with the following syntax:

```
selected-object = RECORD-object "." field-identifier
                | field-designator-identifier.

field-designator-identifier
                = field-identifier.
```

A selected object is a component of an object of a RECORD type (a RECORD-object). The component is specified either by specifying a RECORD-object and the field-identifier or by means of a field designator identifier. Specifying a field designator identifier is only possible as part of a WITH statement (see section 10.5).

If the RECORD-object is a variable access, the selected object is likewise a variable access. The type of the selected object is the type of the RECORD-object component identified by the field-identifier or field designator identifier.

If the RECORD-object is a static expression, the selected object is likewise a static expression.

Fields in the variant part can be accessed only if the corresponding variant has been switched on (active). The values of all components of all variants are totally undefined so long as no variant is active (see also section 7.3). If a new variant is activated, all values of components in the old variant are lost, and cannot even be retrieved by reactivating the old variant.

- **Tag field exists**

  Moving a valid value to the tag field activates the variant which is linked to that value. From this point on, it is possible to access the components belonging to this variant. The values of all components, however, are still totally undefined at this point in time, unless the variant was activated immediately beforehand.

  If the tag field does not have a defined value, no variant is activated and all component values of all variants are totally undefined.

- **No tag field exists**

  Accessing a component of a variant (for reading or writing) activates the variant. At this point in time the components of the variant are still totally undefined, unless the variant was activated immediately beforehand. Thus, converting a variant to a component by means of read access leads to the error of reading an undefined value.

Possible runtime errors:

`Variant_Error`	`- A non-active variant of a RECORD-object is accessed although the variant has a tag field.`

`unpredictable effects`	`- The variant of a RECORD-variable is not active for the total duration of each reference to each of its components.`
	`- In an identified variable created with New(p, c1, .., cn) or New(p, c1, .., cn, e) a variant is activated other than the one determined by the CASE constants c1 to cn.`

Notes

– The frequent practice of assigning values to the components of a variable, then activating another variant and again reading the values of the components (under different types) is illegal. Type conversion of this sort should, if necessary at all, be carried out with the aid of the required function Convert (see section 15.10).

– Access of a non-active variant can only be detected with the Check option if a tag field exists for this variant part.

Example

```
VAR
   person : RECORD
              lastname,
              firstname : PACKED ARRAY [1..20] OF Char;
              age       : 0..100;
              birthday  : RECORD
                            year : 0..2000;
                            month: 1..12;
                            day  : 1..31;
                          END;
            END;
```

The following selected objects are component variables of "person":

`person.lastname, person.firstname, person.age, person.birthday.`

"year", "month" and "day", on the other hand, are components of "person.birthday", not of "person". This is indicated as follows:

```
person.birthday.year
person.birthday.month
person.birthday.day
```

Example of an object which is not a variable access

```
TYPE
  complex = RECORD
               real_part,
               imag_part : Real
            END;

FUNCTION add_complex (x,y : complex) : complex;
BEGIN
  add_complex := complex (x.real_part + y.real_part,
                          x.imag_part + y.imag_part);
END;

VAR
   x,y : complex;
   r   : Real;

BEGIN
    ...
    r := add_complex (x,y).real_part; ——————————— (1)
   ...
END
```

(1) The function call add_complex (x, y) returns a RECORD-type value as a result. The selected object add_complex (x, y).real_part is not a variable access, since it is the component of a function result and not of a variable.

Cross-references

RECORD type:	6.3.3
Defined variable:	7.3
WITH statement:	10.5
Scope:	12
Convert:	15.10
Check option:	16

Dereferenced Objects

```
dereferenced-object = pointer-object "↑" .
```

If the value of the Pointer-object is defined in a dereferenced object and is not NIL (i.e. is a pointer to an identified variable), the dereferenced object stands for this identified variable.

In Pascal-XT, the Pointer-object can be a constant, a variable, an aggregate component or a function call. The dereferenced object, however, is always a variable access.

In Standard Pascal, on the other hand, only variable accesses are permitted as Pointer-objects.

The type of the Pointer-object must be a Pointer type which is neither the generic pointer type (see section 6.5.2) nor a private pointer type of a foreign pak-kage (see section 11.2). The type of the dereferenced object is the domain type of the type of the Pointer-object.

In no case is the dereferencing of a Pointer-object a static expression.

Possible runtime errors:

Pointer_Error	- In a dereferenced object, the value of the *Pointer*-object is NIL.
unpredictable effects	- In a dereferenced object, the value of the *Pointer*-object is undefined. - An identified variable created using New(p,c1,...,cn), New(p,e) or New(p,c1,...,cn,e) occurs as a whole (i.e. not merely some of its components) as an operand in an expression.

Notes

– Let p be a Pointer variable. A reference to the identified variable p↑ is established with a passed variable parameter, in a WITH statement (WITH p↑ DO ...) or with an assignment (p↑ := ...). It is illegal to execute Dispose (p) in subprograms with parameter p↑ or within WITH statements. In assignments, a function which executes a Dispose (p) as a side effect is not allowed on the right-hand side.

– An identified variable specified on the heap in abbreviated form must not occur as a whole in an expression, e.g. on the right-hand side of an assignment. This would cause the variable to be accessed in its entire size (corresponding to its type), including those ranges which were not even assigned to the variable due to its abbreviated form.
  This restriction does not apply to identified variables of the variable string type as they are always maintained at their actual length.

Examples

Two in-depth examples can be found in section 18.3.

```
p↑.firstname              { simple dereferencing }

p↑.spouse↑.firstname      { dereferencing a selected object }

f(x,y,z)↑.firstname       { dereferencing a function call }
```

Cross-references

Pointer types:	6.4
NIL:	6.4
Generic pointer type:	6.5.2
Identifier variable:	7.2
Functions:	8.2
Private pointer type:	11.4
New, Dispose, Release:	15.2

Buffer Variables

```
buffer-variable = FILE-object "↑".
```

A FILE-object (FILE-variable) is a generalized FILE-type variable. For each FILE-variable a buffer variable f↑ is implicitly declared. A buffer variable belonging to a textfile has the type Char; otherwise, the type of the buffer variable is the component type of the FILE type.

The type of the FILE-object must not be the generic FILE type Any_File.

Possible runtime errors:

unpredictable effects	- The file pointer of a file variable f is modified (e.g. by reading or writing) although there still exists a reference to the buffer variable f↑.

Note

Variables of the generic FILE type Any_File cannot be declared and therefore do not have a buffer variable.

Examples of access to buffer variables in conjunction with Read and Write can be found in section 15.1 and in chapter 19.

So long as a reference to the buffer variable of a file variable exists, the value of the file variable (especially the file pointer) must not be changed.
Let f be a FILE variable. A reference to the buffer variable f↑ is established by passing a variable parameter (p (.., f↑, ..)), in a WITH statement (WITH f↑ DO ...) or with an assignment (f↑ := ...). The value of the FILE variable f is changed, for example, when the file pointer is incremented with Put (f) or Get (f) (and hence also with Read and Wri-te) within the called subprogram or WITH statement. Similarly, a function could illegally increment the file pointer as a side effect on the right-hand side of an assignment.

Examples

```
VAR
   i       :  Integer;
   s       :  String;
   data    :  FILE OF Integer;
   outputs :  ARRAY [1..3] OF Text;
   club    :  FILE OF person;          { see section 6.3.3 }

BEGIN
   ...
   data↑ := 1;
   i     := data ↑;

   outputs [i] ↑ := ’x’;

   club↑.lastname := pascal;
   club↑ := person (’blaise’, ’pascal’, 50, False);
END
```

Cross-references

FILE type:	6.3.5
Textfile:	6.3.5.2
Generic FILE type:	6.5.1
Buffer variable:	7.3, 15, 16.1
Variable parameter:	8.5.2
WITH statement:	10.5
Read, Write:	15.1

Statements

Statements represent the algorithm of the program. Any statement (including the empty one) may be preceded by a label. In Pascal, statements are separated by semicolons. A statement has to be followed by a semicolon only if another statement follows. Compound statements, conditional statements, repetitive statements and the WITH statement all contain one or more statements within them.

```
statement  = [label ":"]
             (   simple-statement     | conditional-statement
               | repetitive-statement | compound-statement
               | WITH-statement ).
```

A statement preceded by a label can be branched to with the GOTO statement.

Statement class	Statements	Section	Remarks / Examples
simple statements	empty statement	10.1.1	
	assignment	10.1.2	a := b + c
	procedure call	10.1.3	plus (erg, b, c)
	GOTO statement	10.1.4	GOTO 4711
	EXIT statement	10.1.5	Leave the surrounding repetitive statement
	RETURN statement	10.1.6	Leave the surrounding block (e.g. subprogram)
conditional statements	IF statement	10.3.1	IF bed THEN ... ELSE ;
	CASE statement	10.3.2	multiple selection
repetitive statements	REPEAT statement	10.4.1	REPEAT ... UNTIL exp
	WHILE statement	10.4.2	WHILE exp DO ...
	FOR statement	10.4.3	FOR i:=a1 TO b1 DO ...
compound statement		10.2	BEGIN ... END
WITH statement		10.5	WITH rec_var DO ...

Table 10-1 Statements

Cross-references

Labels: 3.6, 4

Simple Statements

A simple statement is a statement that does not contain any further statements.

```
simple-statement    = empty-statement | assignment
                    | procedure-call  | GOTO-statement
                    | EXIT-statement  | RETURN-statement.
```

Empty Statement

An empty statement does not perform any actions. It is used e.g. to put a jump label in front of END.

```
empty-statement =.
```

Note

It is not necessary, but thoroughly useful, to end statements before END with a semicolon. The fact that this inserts an empty statement between the semicolon and the END has no effect whatsoever on the code generation.
This has the following advantages:

– All statement end with a semicolon.
– When a statement is inserted before END, it is no longer necessary to add a semicolon to the preceding one.

Caution:
A semicolon is never permitted before the ELSE of an IF statement!

Examples

I := 1;;;;

In the above example, each pair of consecutive semicolons contains an empty statement (invisible, of course). Thus, the line in the example contains a total of three empty statements.

```
BEGIN
   I := 5;
   J := 2;
END;
```

In this example, the keyword END is preceded by an empty statement (due to the semicolon before "2"; see Note).

```
     BEGIN
        ...
        GOTO 13;
        ...
  13:END;
```

In this example, "13:" is a marked empty statement. This is because jump labels can only mark statements, not END.

```
     CASE i OF
        1: Writeln ('Special case 1');
        2: Writeln ('Special case 2');
        ELSE:
     END;
```

In this example, the CASE statement contains an empty ELSE branch, i.e. an ELSE branch containing an empty statement. This causes those values of i which are neither 1 nor 2 to be ignored (i.e. not to cause a Case_Error).

```
     FOR i := 1 TO 10000 DO;

     FOR i := 1 TO 10000 DO BEGIN END;
```

Each of the two FOR statements shown above does "nothing" 10000 times, since there is an empty statement between DO and semicolon in the one case and between BEGIN and END in the other. (Whether these "empty loops" really are run through 10000 times is left to each Pascal-XT implementation (code optimization). Caution is therefore advised when using "active wait loops".)

Assignments

Assignments have the following general form:

```
assignment  = variable-object ":=" expression
            | function-identifier ":=" expression.
```

An assignment transfers the value resulting from the evaluation of the expression on the right-hand side of the assignment to the variable access (variable-object) or the function-identifier on the left-hand side of the assignment. The value of the expression must be assignment compatible (see section 6.6.3) with the type of the variable access or the function-identifier. The assignment to the function-identifier must be contained in the associated function-block (see also section 8.2).

A variable or a function result is undefined if it has not yet been assigned a value (see also section 7.3).

Implementation-dependent characteristic

The sequence in which the left-hand and right-hand sides of an assignment are evaluated is implementation-dependent. The reference obtained when the variable-object is evaluated (left-hand side) is retained during the entire time the statement is being executed. This is especially important in connection with possible runtime errors (see sections 9.6.2, 9.6.3, 9.6.4, 9.6.5).

Possible runtime errors:

Range_Error	- In an assignment, the value of an ordinal-type expression (right-hand side) does not lie in the value range of the *ordinal*-type of the variable access or the *function*-identifier (left-hand side).
Numeric_Error	- In an assignment, the value of the expression (right-hand side) of type Long_Real does not lie in the value range of the variable access or the *function*-identifier (left-hand side) of type Short_Real.
Set_Error	- In an assignment, the value of a SET-type expression (right-hand side) does not lie in the value range of the SET type of the variable access or the the *function*-identifier (left-hand side).
String_Error	- In an assignment, the actual length of the character string value of the expression (right-hand side) is greater than the maximum length of the variable string type of the variable access or *function*-identifier (left-hand side).
	- In an assignment, the actual length of the character string expression of a variable string type (right-hand side) is not equal to the length of the fixed character string type of the variable access or the *function*-identifier (left-hand side).
unpredictable effects	- In an assignment, the type of the expression (right-hand side) of a generic pointer type and the Pointer value of the expression point to an identified variable whose type differs from the domain type of the variable access or the f*unction*-identifier (left-hand side).

Examples of assignments to variables

```
i       := 0;
a [50] := 100;
field [x + y] := pi;
b := (1 < i) AND (i < 100);
p↑.age := 3;
```

Example of assignments to function identifiers

```
FUNCTION max (x, y : Integer);
BEGIN
   IF x > y THEN max := x ELSE max := y;
END;
```

Notes

– For functions with a structured result type, the function-identifier must be assigned a value as a whole. Only aggregates are allowed to have values assigned to their components (see section 9.5).

– The assignment of a value to a function-identifier should be located directly in the statement part of the function. It is equally possible to place the assignment within a nested function or procedure, but this is less easy to read.

– The implementation-dependent access sequence to the variable access and the evaluation of the expression may lead to conflicting results.

– A Numeric_Error can only occur for an assignment if the types Short_Real and Long_Real have different value ranges.

Example of an implementation-dependent execution sequence

```
VAR
   i : Integer;
   a : ARRAY [1 .. 9] OF Char;

FUNCTION function_with_side_effect:Char;
BEGIN
   i := 8; { ←- This is the side effect. }
   function_with_side_effect := ’X’ ;
END;

BEGIN
   i    := 2;
   a[i] := function_with_side_effect
END.
```

In the above example, the effect depends on the access sequence to a[i] and the call of function_with_side_effect.

If the reference to a[i] is established first, after which the function is called, then the character ’X’ is assigned to the component a[2].

If, on the other hand, the function is called first, the character ’X’ is assigned to the component a[8].

As mentioned in section 1.4, programs whose effects depend on implementation-dependent characteristics are considered errored.

Cross-references

Implementation-dependent:	2.2
Ordinal type:	6.2
Real type:	6.2.2
Generalized string type:	6.3.2
Variable string type:	6.3.2.2
SET type:	6.3.4
Pointer type:	6.4
Generic pointer type:	6.5.2
Assignment-compatibility:	6.6.3
Variable access:	7
Function result:	8.2
Side effects:	8.2
Expression:	9
Aggregates:	9.5
Object:	9.6
Statement part:	12.1

Procedure Calls

Procedure calls (also known as procedure statements) have the following syntax:

```
procedure-call  = procedure-name [actual-parameter-list].

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

A procedure call causes the block of the called procedure to be executed (see section 8.7).

If the procedure has formal parameters, it must contain an actual parameter list with the actual parameters.
By specifying a package-identifier in the procedure-name, a procedure of the corresponding package is called.

In a procedure call, the runtime errors described in sections 8.5.1, 8.5.4 and 8.7 may occur when the parameters are passed. Runtime errors may also occur when the statements in the procedure-block are executed, and unless they are handled in the procedure they may be propagated to the call position (see chapter 14).

Example

```
proc1;

trans (a, x, y);

numerics.bisect (fct (x + y), 2.0, 100, y);

Writeln ('The result is ', a * b :10:5);
```

Cross-references

Actual and formal parameters:	8.5
Subprogram call:	8.7
Block:	12.1
Exception handling:	14

GOTO Statement

The GOTO statement has the following syntax:

```
GOTO-statement  = "GOTO" label.
```

A GOTO statement causes program execution to continue from the point indicated by the label in the GOTO statement.

The use of a label in a GOTO statement is permitted only if the label was declared in a label declaration part and if it is located in front of a statement on the same or a higher statement level. In other words, it is prohibited to jump inside a subprogram or a structured statement from the outside. Nor is it permitted to jump outside the bound(s) of alternative branches within a structured statement with alternatives (IF, CASE).

Example

```
GOTO 4711;
GOTO 1;
```

Note

GOTO statements should only be used sparingly. Most algorithms can be programmed without them due to the fact that the conditional, repetitive,
EXIT and RETURN statements are available for structured control of the
program flow.

Cross-references

Labels:	3.6
Label declaration part:	4
Labels before statements:	10
Block:	12.1

EXIT Statement

```
EXIT-statement  = "EXIT".
```

This statement may only occur inside a repetitive statement (i.e. a WHILE, REPEAT or FOR statement). In the case of nested repetitive statements, the only statement terminated is the innermost repetitive statement which closest-contains the EXIT statement.

Examples

```
FOR i:= 1 TO n DO BEGIN
   IF a[i] = 0 THEN EXIT;
   s := s + a[i];
   END;
Writeln (s);
```

This example adds up the values of the components of ARRAY "a" up to the first component with the value 0.

In the example below, each input line processes the characters of a line up to the first semicolon; the remaining characters in the line are ignored.

```
Reset;
WHILE NOT Eof DO BEGIN
   WHILE NOT Eoln DO BEGIN
      Read (c);
      IF c = ';' THEN EXIT;
      processing (c);
      END;
   Readln;
   END;
```

Cross-references

Repetitive statement:	10.4

RETURN Statement

```
RETURN-statement = "RETURN" .
```

A RETURN statement causes execution of the block closest-containing it to terminate immediately. A RETURN statement in the statement part of the main program block causes termination of the program.

Note

Before leaving a function-block, the function-identifier must be assigned a value; otherwise, the function result will be undefined.
undefined.

Example

```
FUNCTION find (x : Integer) : Integer;
VAR i : Integer;
BEGIN
   FOR i := 1 TO 100 DO
      IF a[i] = x THEN BEGIN
         find := i;
         RETURN; {found}
         END;
   find := 0; {not found}
END;
```

In this example, RETURN not only terminates the FOR statement but also (unlike EXIT) execution of the function block. Thus, the statement find := 0; is left unexecuted and the function result is find := i, the located index.

Cross-references

Function:	8.2
Block:	12.1

Compound Statements

Compound statements have the following general form:

```
compound-statement = "BEGIN" statement-sequence [EXCEPTION-part] "END".

EXCEPTION-part     = "EXCEPTION" statement-sequence.

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

A compound statement unites one or more statements syntactically into a single statement. This is necessary wherever only one statement is permitted but a more lengthy statement sequence is required (e.g. in conditional statements). The statements in the statement sequence are usually executed in the order in which they are written.

Execution of the statement sequence may be interrupted by executing a GOTO statement (see section 10.1.4), an EXIT statement (see section 10.1.5), or a RETURN statement (see section 10.1.6). It can also be interrupted by calling the required procedure Raise (see section 15.11) or by the occurrence of an exception condition.

Within a compound statement, an exception handling part (EXCEPTION part) may also be defined. This part can be used to handle exceptions occurring when the statement sequence of the compound statement is executed (see chapter 14).

The statement sequence in the EXCEPTION part, assuming this part exists, will only be executed if an exception situation occurs.

Examples

```
BEGIN
   z := x;
   x := y;
   y := z;
END;

BEGIN
   erg := a * b;
EXCEPTION
   Writeln ('Overflow');
   erg := 0;
END;
```

Cross-references

GOTO statement:	10.1.4
EXIT statement:	10.1.5
RETURN statement:	10.1.6
Exception handling:	14
Raise:	15.11

Conditional Statements

Conditional statements have the following general form:

```
conditional-statement  = IF-statement | CASE-statement.
```

The conditional statements include the IF statements and the CASE statements. Depending on the value of an expression, particular statements in the interior of the conditional statement are executed or skipped.

IF Statement

IF statements may be written with or without an ELSE part:

```
IF-statement  = "IF" Boolean-expression "THEN" statement
                [ELSE-part].

ELSE-part     = "ELSE" statement.
```

When an IF statement is executed, the first thing that happens is the evaluation of the Boolean-expression. If this has the value True, the statement after THEN will be executed; otherwise, the statement in the ELSE part (if existent) will be executed.

If, instead of a single statement, an entire statement sequence is to be executed, the statements must be combined into a compound statement.

IF statements may be nested to any depth. If the statement after THEN is followed by ELSE, the ELSE is always assigned to the innermost unterminated IF statement which does not have an ELSE part (see example).

ELSE must never be preceded by a semicolon. A semicolon after the statement following the keyword THEN terminates the IF statement.

Examples

```
IF a > maximum THEN
   maximum := a;

IF a > maximum THEN
   maximum := a
ELSE IF a < minimum THEN
   minimum := a;

IF j = 0 THEN
   IF i = 0 THEN
      Writeln ('undefined')
   ELSE
      Writeln ('infinite')
ELSE
   Writeln (i / j);

IF a = b THEN
   IF c = d THEN
      x := x + 1
   ELSE
      y := y + 1
```

In this example, the ELSE part belongs to the inner IF statement (IF c = d ...), i.e. the statement y := y + 1 will be executed if a = b and c <> d are true.

If, on the other hand, the ELSE part is to belong to an outer IF statement (IF a = b ...), i.e. the statement y := y + 1 is to be executed when a <> b, then one of the following two notations must be used.

```
IF a = b THEN BEGIN
   IF c = d THEN
      x := x + 1
   END
ELSE
   y := y + 1
```

Here the inner IF statement is terminated by the end of the surrounding compound statement (BEGIN ... END). As a result, only the external IF statement is "open" and the ELSE part is therefore assigned to it.

```
IF a = b THEN
   IF c = d THEN
      x := x + 1
   ELSE
      {in this case nothing happens,}
      {there is an empty statement here}
ELSE
   y := y + 1
```

Here the inner IF statement is terminated by a separate ELSE branch (containing an empty statement). The second ELSE branch (with the statement y := y + 1) will therefore be assigned to the outer IF statement.

Unlike a CASE statement, it is irrelevant for the effect of an IF statement whether it contains an ELSE part with an empty statement or no ELSE part at all. The above example, however, illustrates how important an "empty" ELSE part may be for the context.

Cross-references

Boolean:	6.2.4
Expression:	9
Empty statement:	10.1.1
Compound statement:	10.1.2

CASE Statement

CASE statements have the following general form:

```
CASE-statement  = "CASE" case-index "OF" case-list [";"] "END".

case-list       = ordinal-expression.

case-list       = case-list-element {";" case-list-element}.

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

selector-list   = selector {"," selector} | "ELSE".

selector        = case-constant [".." case-constant] .

case-constant   = ordinal-constant.
```

The CASE statement is used to execute a statement selected from a list of alternative statements by means of an expression.

The expression in the case index must be of an ordinal type. The CASE constants of the selector lists must be values of this ordinal type. A selector in the form a..b stands for an enumeration of all constants c such that a ≤ c ≤ b. The individual value ranges must differ pair-by-pair.

The selector list of the last case list element may be the word symbol ELSE. This case list element is called the ELSE part. ELSE then stands for the values of the case index type which did not occur in the previously executed selector lists. If all values of the case index type occur in the selector lists, the ELSE part has no effect.

When the CASE statement is entered, the first thing that happens is that the case index is computed. If the result is contained in a selector list, the associated statement will be executed.
If it is not contained in a selector list, the statement of the ELSE part will be executed, provided that an ELSE part exists.

Note

Unlike the IF statement, there is a semicolon in front of the ELSE in a CASE statement.

Possible runtime errors:

`Case_Error`	`- In a CASE statement, there is no CASE constant corresponding to the value of the case index,`

```
                    nor is an ELSE part specified.
```

Examples

```
CASE character OF
   '0'..'9': z := Ord (character) - Ord ('0');
   '+'     : z := x + y;
   '-'     : z := x - y;
   '*'     : z := x * y;
   END;
```

A runtime error (Case_Error) occurs if "character" assumes a value other than '+', '-', '*' or a digit.

```
CASE character OF
   '0'..'9': z := Ord (character) - Ord ('0');
   '+'     : z := x + y;
   '-'     : z := x - y;
   '*'     : z := x * y;
ELSE       : Writeln ('wrong input');
END;
```

The statement in the ELSE part is executed for all values of "character" other than '+', '-', '*' or a digit.

```
CASE character OF
   '0'..'9': ...;
   ...:      ...;
   '*':      ...;
   ELSE:        ; {in this case nothing happens}
   END;
```

All values of "character" other than '+', '-', '*' or a digit are ignored (because the ELSE part contains an empty statement).

A comparison of the first example with the last one shows that, unlike an IF IF statement, it is by no means irrelevant to the effect of a CASE statement whether it contains an ELSE part with an empty statement or no ELSE part at all. "Wrong" values of the case index always trigger an action of some sort, but may in other cases lead to a runtime error (detected as a Case_Error if Check=On).

Cross-references

Constant:	5
Expression:	9
Ordinal expression:	9
Empty statement:	10.1.1

Repetitive Statements

Repetitive statements are:

```
repetitive-statement  = REPEAT-statement
                      | WHILE-statement
                      | FOR-statement.
```

Repetitive statements cause those statements contained within them (the "inner" statements) to be executed repeatedly. REPEAT statements and WHILE statements are condition-controlled repetitive statements. FOR statements are controlled by means of a control variable whose initial and final values are defined. Repetitive statements may be abandoned with the EXIT statement.

REPEAT Statement

REPEAT statements have the following general form:

```
REPEAT-statement  = "REPEAT" statement-sequence
                    "UNTIL" Boolean-expression.
```

The statement sequence in the REPEAT statement is executed repeatedly until the Boolean expression after UNTIL returns the value True. The statement sequence is executed at least once, since the condition is not evaluated until after the statement sequence has been processed.

Statement sequence execution may be abandoned by executing a GOTO statement (see section 10.1.4), an EXIT statement (see section 10.1.5) or a RETURN statement (see section 10.1.6), or by calling the required procedure Raise (see section 15.11), or by the occurrence of an exception situation.

Examples

The example below implements Euclid’s algorithm for calculating the greatest common divisor (GCD) of two integers i and j by means of the MOD operator (see section 6.3.1). The "divisor" of MOD must never be less than zero; this must be ascertained at the beginning of this statement sequence. Following the last program run, the greatest common divisor of the initial values i and j is located in i.

```
{prerequisite: j > 0}
REPEAT
   k := i MOD j;
   i := j;
   j := k;
UNTIL j = 0;
```

The REPEAT statement is well-suited for implementing user-controlled termination in a loop. For example:

```
REPEAT
   { start of processing }
   ...
   { end of processing }
   Writeln(’Continue? Y/N’);
   Readln;
   Read(character);
UNTIL character IN [’N’,’n’];
```

If it is not clear from the outset whether processing is to take place at all, the WHILE statement is more appropriate.

Cross-references

Boolean:	6.2.4
Expression:	9
GOTO statement:	10.1.4
EXIT statement:	10.1.5
RETURN statement:	10.1.6
Statement sequence:	10.2
Exception handling:	14
Raise:	15.11

WHILE Statement

WHILE statements have the following general form:

```
WHILE-statement = "WHILE" Boolean-expression "DO" statement.
```

If the Boolean expression in a WHILE statement returns the value True, the statement contained within the WHILE statement (the "inner" statement) is executed and repeated until the Boolean expression returns the value False. If a sequence of statements is to be repeated, they must be combined into a compound statement.

Execution of the inner statement can be interrupted by executing a GOTO statement (see section 10.1.4), an EXIT statement (see section 10.1.5) or a RETURN statement (see section 10.1.6), or by calling the required procedure Raise (see section 15.11), or by the occurrence of an exception situation.

Example

The example below implements Euclid's algorithm for the greatest common divisor (GCD), as already described for the REPEAT statement (see section 10.4.1).

```
WHILE j > 0 DO BEGIN
   k := i mod j;
   i := j;
   j := k;
   END;
```

In the next example, two numbers are read from every input line of file f and processed until the end of the input file is reached. Due to the use of the WHILE statement (in contrast to the REPEAT statement), even empty input files are handled correctly.

```
Reset(f);            { fetch 1st line or recognize EOF }
WHILE NOT Eof(f) DO BEGIN
   Read(f,i);        { supply i and k from current line }
   Read(f,k);
   process (i, k);   { processing }
   Readln(f);        { fetch new line or recognize EOF  }
   END;
```

Cross-references

Boolean:	6.2.4
Expression:	9
GOTO statement:	10.1.4
EXIT statement:	10.1.5
RETURN statement:	10.1.6
Exception handling:	14
Raise:	15.11

FOR Statement

FOR statements have the following general form:

```
FOR-statement     = "FOR" control-variable ":=" initial-value
                    ( "TO" | "DOWNTO" ) final-value "DO" statement.

control-variable  = variable-identifier.

initial-value     = ordinal-expression.

final-value       = ordinal-expression.
```

The FOR statement causes the statement contained within it (the "inner" statement) to be executed repeatedly. In doing this, the control variable is assigned, after each loop cycle, a new value from the sequence of successive values between and including the initial and final values (ascending in the case of TO, descending in the case of DOWNTO).

Execution of the inner statement may be interrupted by executing a GOTO statement (see section 10.1.4), an EXIT statement (see section 10.1.5) or a RETURN statement (see section 10.1.6), or by calling the required procedure Raise (see section 15.11) or by the occurrence of an exception situation.

The control variable must be an entire variable whose identifier is defined in the variable declaration part of the block closest-containing the FOR statement. This means that global variables are not permitted as control variables. The control variable must be of an ordinal type, and the type of the initial and final values must be compatible with this type.

The initial and final values of the FOR statement are only evaluated prior to the first loop cycle. As a result, the loop cannot be abandoned by changing the final value.

If the initial value is greater than the final value (or less than in the case of DOWNTO), the inner statement is not executed at all; if they are equal to each other, the statement is executed exactly once.

If the inner statement is executed at least once, the initial and final values must be assignment-compatible with the type of the control variable.

Once the FOR statement has been executed, the value of the control variable is undefined, unless the FOR statement has been exited by means of a GOTO or EXIT statement. In this case, the control value has its current actual value. "Undefined" here means in particular that one cannot be sure that the control variable will have the final value after the FOR statement has been exited.

Neither the FOR statement nor a procedure or function declaration in the block closest-containing the FOR statement may contain a statement which threatens the control variable (i.e. which can potentially change its value).

A statement S threatens an ordinal-type variable V if one of the following propositions is true:

a) S is an assignment and V is its left-hand side.

b) S is a procedure call or contains a function call in which V occurs as an actual parameter, and the associated formal parameter is a variable parameter.

c) S is a call for the required procedure Read, Readln or Readstring, and V is a parameter in S.

d) S is a FOR statement and V represents the control variable in S.

Possible runtime errors:

`Range_Error`	`- When the statement in a FOR statement is executed, the initial or final value of the FOR statement does not lie in the value range of the type of the control variable.`

Example

```
FOR i := 1 TO 100 DO
   FOR j := 1 TO i-1 DO BEGIN
      h := a [i, j];
      a [i, j]: = a [j, i];
      a [j, i] := h;
      END;
```

Notes

– The value of a control variable must not be changed (threatened) by the user. This can also happen in a nested subprogram called in the FOR loop. The call may be dependent on data which only become known at execution time. For this reason, the restrictions have been extended to include all nested subprograms.

– FOR statements are particularly well-suited for processing arrays, since with arrays the number of elements in a line, column etc. is known in advance. As the example shows, the initial and final values of the control values need not be constant, but they must have defined values at the moment FOR statement execution begins.

Cross-references

Ordinal type:	6.2
Compatible:	6.6.2
Assignment-compatible:	6.6.3
Subprogram call:	8.7
Expression:	9
GOTO statement:	10.1.4
EXIT statement:	10.1.5
RETURN statement:	10.1.6
Exception handling:	14
Read, Readln:	15.1, 19
Readstring:	15.3

WITH Statement

WITH statements have the following general form:

```
WITH-statement  = "WITH" RECORD-variable-list "DO" statement.

RECORD-variable-list
                = RECORD-variable-object {"," RECORD-variable-object}.
```

The WITH statement opens the scope of one or more RECORD-variables so that the field identifiers of these variables can be accessed directly in the "inner" statement without specifying the RECORD-variables (see section 9.6.2).

The identifiers, now made visible, conceal all other like-named identifiers defined further outside. Particularly in the case of nested WITH statements, the scope rules should always be kept in mind.

The RECORD-variables are accessed at the beginning of WITH statement execution, and references to the variables are retained for the entire time the WITH statement is being processed.

This is important for two reasons:

1. It may involve certain runtime errors (see sections 9.5.2, 9.5.3, 9.5.4 and examples 2, 3 and 4).
2. If a RECORD-variable in a WITH statement contains an indexed object or pointer dereferencing, changes made to the index expressions or Pointer-objects while the inner statement of the WITH statement is being executed no longer affect the reference (see examples 5 and 6).

The statement

```
WITH v1, v2, ..., vn DO statement
```

is equivalent to

```
WITH v1 DO
   WITH v2 DO
      ...
         WITH vn DO statement
```

Notes

- WITH statements are usually used to reduce the effort of writing. This happens, of course, at the expense of readability.
- If two or more RECORD-variables have like-named components, nested WITH statements pose a potential danger since field designator identifiers mutually overlap. It is not always immediately apparent which component of which RECORD-variable is being addressed.

Example 1

By means of a WITH statement, the statement

```
IF date.month = 12 THEN BEGIN
   date.month := 1;
   date.year  := date.year + 1;
   END
ELSE
   date.month := date.month + 1;
```

can be abbreviated as follows:

```
WITH date DO
   IF month = 12 THEN BEGIN
      month := 1;
      year  := year + 1;
      END
   ELSE
      month := month + 1;
```

Example 2

The next example illustrates an error in the use of a WITH statement. The value (namely, the file pointer) of the FILE-variable f is modified by Get(f) although there is still a reference to the buffer variable f↑ due to the WITH statement.

```
VAR
   f : FILE OF record_type;
BEGIN
   ...
   WITH f↑ DO BEGIN
      ...
      Get (f); {this is an error with unpredictable effects}
      ...
      END;
   ...
   END
```

Example 3

The example below illustrates another error in the use of a WITH statement. Dispose (p) removes an identifying value although there is still a reference to the identified variable p↑ due to the WITH statement (see section 9.5.3).

```
VAR
   p : ↑record_type;
BEGIN
   New (p);
   WITH p↑ DO BEGIN
      ...
      Dispose (p); {this is an error with unpredictable effects}
      ...
      END;
   ...
END.
```

Example 4

This example illustrates a third error in the use of a WITH statement. As a result of the statement r.b := False, the "True variant" of the RECORD variable r is not active for the entire duration of the reference to its component r.y, even though this reference exists due to the WITH statement (see section 9.5.2).

```
VAR
   r :  RECORD
           CASE b : Boolean OF
              False: (x : Integer);
              True:  (y : record_type);
        END;

BEGIN
   r.b := True;
   WITH r.y DO BEGIN
      ...
      r.b := False; {this is an error with unpredictable effects}
      ...
      END;
   ...
END.
```

Example 5

The example below shows a case where modifying an index expression has no effect on the reference of an indexed object.

```
VAR
   i : Integer;
   a : ARRAY [1..10] OF RECORD x : ... END;

BEGIN
   i := 2;
   WITH a[i] DO BEGIN {establishes reference to a[2] (=a[i])}
      ...
      i := 8; {reference to a[2] remains in effect}
      ...
      x := ...; {still refers therefore to a[2].x
                 but not to a[8].x (=a[i].x)}
      ...
      END;
   ...
END.
```

Example 6

This final example shows the case where modifying a Pointer object has no effect on the reference to the identified variable.

```
TYPE
   record_ptr  = ↑record_type;
   record_type = RECORD
                   x,
                   y    : Integer;
                   next : record_ptr;
                 END;

VAR
   p,
   list : record_ptr;

BEGIN
   ...
   p := list;
   WITH p↑ DO BEGIN {establishes reference to list↑ (=p↑) }
      WHILE (p <> NIL) AND THEN (x > 0) DO BEGIN
         process (x, y);
         p := next; {reference to list↑ remains in effect}
         END {WHILE};
      END {WITH};
   ...
END.
```

This program part does not have the desired effect (i.e. accessing the components of the dynamic RECORD variables list↑, list↑.next↑, list↑.next↑.next↑, ...). The reason is that during the entire time the WITH statement is being executed (and thus during all loop cycles of the WHILE statement) the reference to list↑ (which was set up at the beginning with the WITH statement) is retained although the Pointer variable p is changed by p := next. All accesses to x, y and next thus refer again and again only to list↑.x, list↑.y and list↑.next.

The statement part should thus be written correctly as follows:

```
BEGIN
   ...
   p := list;
   WHILE (p <> NIL) AND THEN (p↑.x > 0) DO BEGIN
      WITH p↑ DO BEGIN
         process (x, y);
         p := next;
         END {WITH};
      END {WHILE};
   ...
END.
```

Cross-references

Field designator identifier:	6.3.3
RECORD type:	6.3.3
Object:	9.6
Scope rules:	12

Main Program and Packages

A Pascal-XT program consists of one or more compilation units, of which exactly one must be a main program.

In Standard Pascal, a program consists of a single main program only.

Main Program

```
main-program   = {context-specification}
                 "PROGRAM" identifier
                 ["(" program-parameter-list ")"] ";"
                 main-program-block ".".

program-parameter-list
               = identifier-list.
```

A main program is a compilation unit. The identifier following the keyword PROGRAM is the program name. In Pascal-XT, the program name must be different from the identifiers of all packages belonging to a program (see section 13.1).
In Standard Pascal, this identifier has no meaning within the program.

The context specification lists the identifiers of the packages to be accessed in the main program. Context specification is described in section 11.3.

The identifiers in the program parameter list must be different from each other. The parameters are described in detail in section 11.5.

Cross-references

Context specification:	11.3
Program parameters:	11.5
Block:	12.1
Scope rules:	12.2
Program structure:	13.1
Compilation units:	13.2
Program execution:	13.3

Sample program: Labyrinth

The example below illustrates a complete Pascal program to solve the labyrinth problem. This problem crops up in similar form in many areas, e.g. in communications engineering or transportation.

Problem:

Find all the ways out of a labyrinth, from a given starting point to the exit.

The labyrinth itself is described by means of a two-dimensional field:

```
VAR lab : ARRAY[0..n,0..n] OF Char;
```

The walls are represented by ’#’, the passageways by blanks. The path traversed is to be marked with dots.

The search strategy used here is implemented by the procedure "search", which is based on a recursive mechanism. The procedure illustrates the potential of the Pascal programming language which, among other things, makes it possible to formulate powerful algorithms with a small number of language elements.

The following actions are performed by the procedure "search":

- Search for the next free location.
- Delete dots in dead ends.
- Avoid walking in circles.
- Retrace a located exit to the first alternative and search for a new path.
- Terminate the search following the last possible path.

```
PROGRAM labyrinth (Output, lfile);

CONST  n      = 16;
TYPE   labtyp = ARRAY [0..n, 0..n] OF Char;
VAR    lab    : labtyp;
       lfile  : Text;

PROCEDURE read_lab (VAR l : labtyp);
VAR i, k: Integer;
BEGIN
   Reset (lfile);
   FOR i := 0 TO n DO BEGIN
      FOR k := 0 TO n DO
         Read (lfile, l[i,k]);
      Readln;
      END;
END { read_lab };

PROCEDURE write_lab (l : labtyp);
VAR i, k: Integer;
BEGIN
   FOR i := 0 TO n DO BEGIN
      FOR k := 0 TO n DO
         Write (l[i,k]);
      Writeln;
      END;
END { write_lab };

PROCEDURE search (i, k: Integer);
BEGIN
  IF lab[i,k] = ' ' THEN BEGIN   { location free ? }
    lab[i,k] := '.';             { set path dot }
    IF (i MOD n = 0) OR
       (k MOD n = 0)             { exit ? }
    THEN write_lab (lab)         { output path }
    ELSE BEGIN                   { recursive call }
       search (i+1,k);           { counter-clockwise }
       search (i,k+1);
       search (i-1,k);
       search (i,k-1);
       END;
    lab[i,k] := ' ';             { Delete dot. The resolu- }
                                 { tion of the recursion   }
                                 { operation automatically }
                                 { causes the path to be   }
                                 { retraced.               }
    END;                         { IF location free }
END;  { search }

BEGIN { labyrinth }
   read_lab (lab);               { read in the labyrinth }
   search (n DIV 2, n DIV 2)     { start search at }
                                 { mid-point }
END.
```

Packages

Packages make it possible to combine logically related declarations. In the simplest form, packages may contain commonly used constant, type and variable declarations. In the more general case, packages specify related data structures and subprograms which work with these data structures. A detailed description of the use of packages can be found in chapter 17.

A package is divided into a package specification and a package body. These are separate compilation units. The package specification contains the outwardly visible part of the package, while the package body is protected from outside access.

The general form for package specification and package body is as follows:

```
package-specification
                = {context-specification}
                  "PACKAGE" identifier
                  ["(" program-parameter-list ")"] ";"
                  { constant-definition-part
                   | type-definition-part
                   | variable-declaration-part
                   | procedure-heading ";" [directive ";"]
                   | function-heading ";" [directive ";"]
                   | "ENTRY" procedure-heading  ";"
                   | "ENTRY" function-heading ";"
                   | INLINE-procedure-declaration
                   | INLINE-function-declaration
                  }
                  "END" ".".

package-body
                = {context-specification}
                   "PACKAGE" "BODY" package-identifier
                  ["(" program-parameter-list ")"] ";"
                  {  constant-definition-part
                   | type-definition-part
                   | variable-declaration-part
                   | procedure-declaration
                   | function-declaration
                  } statement-part ".".
```

The identifier following the keyword PACKAGE is the name of the package. The same name must also occur as the package-identifier following the keyword BODY in the associated package body (see section 13.1).

For each package specification there must be a package body. This package body may also be empty, i.e. consist solely of

"PACKAGE" "BODY" identifier; "BEGIN" "END" "."

(see section 11.2.2).

The declaration and definition parts in the package specification, followed by the declaration and definition parts and the statement part of the package body, form the package-block. A package-block is set up in the same way as a procedure-, function- or main-program-block.
However, it must not closest-contain any label declarations.
Because the package specification and package body are combined into a package-block, the identifiers from the package specification must not be declared or defined again in the body.

The context specification lists the identifiers of those foreign packages which are to be accessed. The identifiers listed in the package specification are also known in the package body; however, those listed in the package body are not known in the package specification. Context specification is described in section 11.3.

Package specification and package body may have separate program parameter lists. The identifiers of the program parameter list of the package specification must be declared in the package specification and are visible in the entire package; the identifiers of the program parameter list of the package body must be declared in the package body and are only visible there. The program parameters are described in detail in section 11.5.

For each procedure heading or function heading specified in the package specification but lacking a directive (other than Forward), the procedure identification or function identification must be specified in the package body. Only for INLINE subprograms is the declaration specified in the package specification (see also section 11.2.1).

Note

A label declaration part is not allowed directly in the package block, since otherwise a branch could be made from a subprogram to the statement part of the package body although this statement part is only executed once (see section 11.2.2).

Example

```
PACKAGE stack;
PROCEDURE push (x: Integer);
PROCEDURE pop  (VAR x: Integer);
FUNCTION  is_empty: Boolean;
END.
```

```
PACKAGE BODY stack;

VAR
   representation: ARRAY [1..100] OF Integer;
   top_of_stack  : 0..100;

PROCEDURE push (x: Integer);
BEGIN
   top_of_stack := top_of_stack + 1;
   representation [top_of_stack] := x;
END {push};

PROCEDURE pop (VAR x: Integer);
BEGIN
   x := representation [top_of_stack];
   top_of_stack := top_of_stack - 1;
END {pop};

FUNCTION is_empty: Boolean
BEGIN
   is_empty := top_of_stack = 0;
END {is_empty};

BEGIN
   top_of_stack := 0;
END {stack}.
```

Cross-references

Constant definition:	5
Type definition:	6
Variable declaration:	7
Procedure declaration:	8.1
Procedure heading:	8.1
Function declaration:	8.2
Function heading:	8.2
INLINE subprograms:	8.3
ENTRY subprograms:	8.4
Context specification:	11.3
Program parameters:	11.5
Scope rules:	12
Program structure:	13.1
Compilation units:	13.2
Program execution:	13.3
Package concept:	17

Package Specification

All identifiers declared and defined in the package specification may also be used outside the package. Declared variables may also be modified outside the package.

In the case of subprograms, attention must be paid to whether directives or the keywords ENTRY or INLINE were specified:

- **Subprograms without directives**

For procedures and functions with no directive specification, only the heading is specified; the associated identifications are not specified until the package body.

- **Subprograms with directives**

For subprograms with a directive other than Forward, there must be no identification for it in the package body. The subprogram block must be in a different language, i.e. outside the package.

- **ENTRY subprograms**

If a procedure or function heading in a package specification is preceded by the word symbol ENTRY, the procedure or function thus declared may also be called by program components not written in Pascal-XT. In this case, implementation-defined interfaces (see user’s guide) must be adhered to. Before a procedure or function is called from a foreign-language program component for the first time, the package containing the procedure or function declaration will be initialized, as will all non-initialized packages which were imported directly or indirectly by this package in WITH lists (see section 13.3).

Implementation-defined characteristic

ENTRY subprograms may be subject to implementation-defined restrictions.

- **INLINE subprograms**

For an INLINE subprogram (usable outside the package), the package specification must also contain a specification of the associated block, since the subprogram call, which may lie outside the package, is replaced (expanded) by the subprogram block (see also section 8.3).
The specification of the subprogram block may have to include, in the

package specification, further declarations and definitions which would otherwise only be required in the package body. By being thus included, these declarations and definitions are made outwardly visible.

Note

A package specification should only contain (and thus make outwardly visible) those declarations and definitions which are absolutely required by another compilation unit.

Example

This example illustrates implementation of a data type with the possible access operations. For purposes of simplification, the data type "nodes" contains only a single field "class", which can only be accessed via the specified subprograms. By implementing these as INLINE subprograms, the accesses are replaced at the call position by the statements specified in the subprogram blocks.

```
PACKAGE data;

TYPE
   range        = 0..255;
   node_pointer = ↑nodes;
   nodes        = RECORD
                    class : range;
                  END;

FUNCTION new_nodes: node_pointer;

INLINE FUNCTION class_of (pointer: node_pointer): range;
BEGIN
   class_of := pointer↑.class
END;

INLINE PROCEDURE set_class (pointer: node_pointer); i: range);
BEGIN
   pointer↑.class := i;
END;

END.
```

Cross-references

Procedure heading:	8.1
Function heading:	8.2
INLINE subprograms:	8.3
ENTRY subprograms:	8.4

Package Body

For each package specification there must be a package body. If a package specification does not contain procedure and function headings to which procedure or function identifications still have to be specified, the associated package body may be "empty".

Unlike identifiers declared in the package specification, the identifiers declared in the package body are visible only inside the package body, and cannot be used outside the package.

The statement part of the package body is executed once and only once when a program is run. The statement part may contain statements for initializing variables (see example at the end of section 11.2).

Notes

- A common package specification in different programs may have different package bodies which satisfy the various requirements regarding memory or runtime conditions. In the package "stack" (see section 11.2), the stack is implemented by means of the ARRAY variable "representation", and thus has a defined size. An alternative body might implement the stack as a list whose size (theoretically) is unlimited. This would not change anything at the interface (package specification).
- The value of a variable declared in the package body can only be changed in the body; the value of a variable declared in the package specification may also be changed outside the package.
- For enhanced readability, the formal parameter list for the subprograms specified in the package specification may be repeated, as can the result type in the case of functions.

Cross-references

Statements:	10
Program parameter list:	11.5
Program execution:	13.3.3

Context Specification

The context specification sets up the connection to other packages.

```
context-specification
                = WITH-list | USE-list.

WITH-list       = "WITH" package-identifier {","
                         package-identifier} ";".

USE-list        = "FROM" package-identifier "USE"
                     imported-identifier {","
                     imported-identifier} ";".

imported-identifier
                = constant-identifier  | type-identifier
                | variable-identifier  | procedure-identifier
                | function-identifier.
```

WITH List

The WITH list is used to define a relation between the compilation units of a program: if the package-identifier P occurs in the WITH list of the compilation unit U, this means that "U relates to the package specification of P". Each package body automatically relates to the associated package specification.

The relation "compilation unit U relates to the compilation unit V" must be a partial ordering to the set of all compilation units belonging to a program. This requirement excludes the possibility that two specifications can mutually relate to each other. However, it is not prohibited for the body of a package P to relate to the specification of a package Q and, at the same time, for the specification or implementation of Q to relate to the specification of P.

Example

```
PACKAGE P;
 .
 .
 .
END.
```

```
PACKAGE Q;
 .
 .
 .
END.
```

```
WITH Q;
PACKAGE BODY P;
 .
 .
 .
BEGIN
END.
```

```
WITH P;
PACKAGE BODY Q;
 .
 .
 .
BEGIN
END.
```

If the same package-identifier occurs more than once in the same or in different WITH lists of a compilation unit, only the first occurrence is significant and the subsequent ones are ignored. In particular, this applies even when the first occurrence takes place in a WITH list of a package specification and a subsequent occurrence takes place in a WITH list of the associated package body.

The specification of a packa-identifier P in a WITH list of a compilation unit U makes it possible for all identifiers declared directly in the specification of P to be used in U by specifying

package-identifier "." identifier

The identifiers from package P are (fully) qualified by specifying the package-identifier. In a package body, identifiers from the associated package specification are used without prefixing the package-identifier.

In a WITH list, it is not allowed to specify the name of a main program (program name).

Example

In the package body b, the enumerated type "color" is used with its constant-identifiers from package a.

```
PACKAGE a;
TYPE
   color = (red, blue, yellow, green, white, violet, orange);
END.


WITH a;
PACKAGE BODY b;
VAR
   f: set of a.color;
BEGIN
   f := [ a.red, a.blue, a.yellow ];
END { b }.
```

USE List

Imported identifiers from a foreign package which are listed in a USE list may be used without prefixing the package-identifier.

The package-identifier listed in a USE list must also be listed in a preceding WITH list, and the imported identifiers must be declared in the package specification of the named package.

If the identifier listed in a USE list identifies an enumerated type, all constant-identifiers forming the values of the enumerated type are also imported implicitly.

If the same identifier occurs more than once in the same or in different USE lists of a compilation unit, only the first occurrence is significant and the subsequent ones are ignored. In particular, this applies eben though the first occurrence takes place in a USE list of a package specification and a subsequent occurrence takes place in a USE list of the associated package body.

Note

To enhance the readability of a program, USE lists should be employed only in limited cases so that the use of an identifier will clearly indicate the location of its declaration or definition.

Example

The example given in section 11.3.1 looks as follows when a USE list is employed:

```
PACKAGE a;
TYPE
   color = (red, blue, yellow, green, white, violet, orange);
END.


WITH a;
FROM a USE color;    { this also imports the identifiers }
                     { red through orange }
PACKAGE BODY b;
VAR
   f: set of color;
...
BEGIN
   f := [ red, blue, yellow ];
END { b }.
```

Private Types

A Pointer type is called a private type when the Pointer type is defined in a package specification and the domain type is not defined until the associated package body. Only the type-identifier is known outside the package, but not the underlying structure of the domain type.

Pointer dereferencing with a pointer of the private pointer type is not permitted outside this package or in INLINE subprograms declared in this package specification. Identifier variables of this type may only be accessed via access procedures and access functions declared in the package specification. The required procedures New and Dispose may likewise be called for private pointer types only within this package.

Note

With the concept of private pointer types, Pascal-XT supports the concept of "information hiding", i.e. the concealment of details in the package body. This enhances the security of a program, since data structures cannot be inadvertently modified from outside.

Example

```
PACKAGE queue_manager;

TYPE queue = ↑element;
FUNCTION tail (q: queue): queue;

END {queue_manager}.



PACKAGE BODY queue_manager;

TYPE element = RECORD
                  next: queue
               END;
FUNCTION tail (q: queue): queue;
BEGIN
   tail := q↑.next
END {Tail};

BEGIN

END {queue_manager}.
```

Cross-references

Pointer types:	6.4

Domain type:	6.4
Dereferencing:	9.6.4
New:	15.2
Dispose:	15.2

Program Parameters

The identifiers in a program parameter list are called program parameters. Each program parameter, except the required identifiers Input and Output, must be declared directly as FILE-variables in the block of the compilation unit in whose program parameter list it occurs (see also implementation-defined and implementation-dependent characteristics).

The program parameters of all compilation units belonging to a program (see also section 13.1) must differ in pairs.
Since Input and Output have a special meaning as program parameters (see below), they are permitted to be listed as program parameters in different compilation units. They then always denote the same file.

In Pascal-XT, the identifiers Input and Output, when they occur as program parameters, are automatically imported from an indirectly addressable package Input Package or Output_Package which declares these identifiers as Text-type variables. When Input_Package is initialized, Reset (Input) is performed implicitly; when Output_Package is initialized, Rewrite (Output) is performed implicitly.

```
PACKAGE Input_Package (Input);
VAR Input: Text;
END { Input_Package }.

PACKAGE Output_Package (Output);
VAR Output: Text;
END { Output_Package }.
```

In Standard Pascal, Input and Output are variable-identifiers of the required type Text. Only when they occur in the program parameter list do they become visible (unlike the other required identifiers), and only then will they automatically be opened using Reset (Input) or Rewrite (Output) before being accessed for the first time.

This somewhat different interpretation of the required identifiers Input and Output in Pascal-XT and in Standard Pascal does not, however, have any effect on a program.

Implementation-defined characteristics

- For program parameters, the assignment to objects outside the program is implementation-defined.
- According to the Pascal standard, the effect of the required procedures Reset or Rewrite on either of the required textfiles Input or Output is implementation-defined. In Pascal-XT, this characteristic is defined for all implementations: namely, an Open_Error occurs.

Implementation-dependent characteristic

For program parameters whose variables do not have a FILE type, the assignment to objects outside the program is implementationdependent.
In Pascal-XT, a variable cannot be specified in the program parameter list if it does not have a FILE type.

Note

The Pascal-XT implementation cannot check whether the program parameters differ pair-by-pair in all compilation units belonging to a program.

Example

```
PACKAGE reporter (message_output);
VAR
   message_output: Text;
...
END.


PACKAGE BODY reporter (message_texts, Output);
VAR
   message_texts: Text;
BEGIN
...
END.


WITH reporter;
FROM reporter USE message_output;
PROGRAM example (Input, Output, file_input);
VAR
   file input: Text;
BEGIN
...
END.
```

The program parameters in this program, apart from Input and Output, are all different. The file "message-texts" is only known in the package body of "reporter", and it is not possible to access this file from outside the package. On the other hand, it is possible to access the package specification, even from outside (see main program).

Cross-references

Textfile:	6.3.5.2
Scope rules:	12
Reset, Rewrite:	15.1
Input, Output:	6.3.5.2, 11.5, 19, A.2

Scope Rules

Every identifier and every label must have a defining point. Each defining point has a region (see section 12.2) which is part of the program text, and a scope (see section 12.3) encompassing the entire region or only part of it. The defining point of an identifier must be located in front of each place where the identifier is used, except in the case when it is used as a domain type in a Pointer type, or when program parameters are used.

Blocks

```
block =    {   label-declaration-part
             | constant-definition-part
             | type-definition-part
             | variable-declaration-part
             | procedure-declaration
             | function-declaration
           }
           statement part .
```

A block consists of declarations and definitions - known for short as the declaration part - and a statement part. Declarations are used to introduce objects with particular characteristics (values, algorithms). Definitions are used to define objects by means of equating a name with another object. The definition is indicated by an equals sign. The statement part determines the algorithmic actions which are performed when the block is executed.

All identifiers and labels used in a block must be declared or defined beforehand, in so far as they are not predefined (required).

A block closest-containing a label declaration part in which a label label is defined must contain exactly one statement marked with this label. There can be any number of branch statements (GOTO statements) specifying this label.

In Pascal-XT, declarations and definitions can be specified in any sequence;
they may also occur more than once.

In Standard Pascal, the general form for a block is as follows:

```
block =    [ label-declaration-part    ]
           [ constant-definition-part  ]
           [ type-definition-part      ]
           [ variable-declaration-part ]
           {   procedure-declaration
             | function-declaration
           }
           statement-part .
```

Blocks are defined recursively. This is because procedure and function declarations in turn contain a block, in so far as they are not declared to be external subprograms (see Directives). A subprogram thus has in turn a declaration part and a statement part. Accordingly, a program may contain many nested blocks.

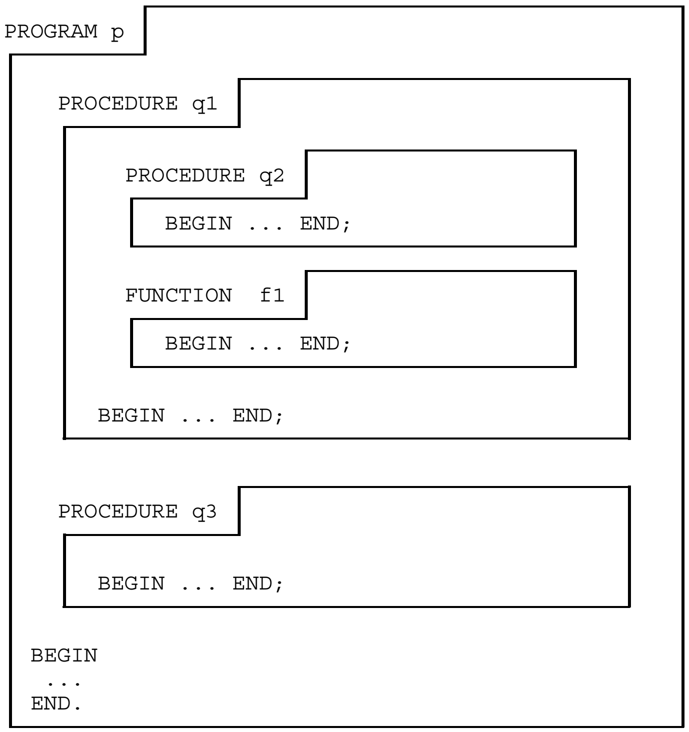


Fig. 12-1 Block structure in a program

Cross-references

Label declaration part:	4
Constant definition part:	5
Type definition part:	6
Variable declaration part:	7
Procedure declaration:	8.1
Function declaration:	8.2
Procedure block:	8.1
Function block:	8.2
Directives:	8.6
Statements:	10
GOTO statement:	10.1.4
Main program block:	11.1
Package block:	11.2
Block:	12.1
Region:	12.2
Defining point:	12.2
Scope:	12.3
Executing a block:	13.3.1
Required identifiers:	A.2

Defining Points and Regions of Identifiers and Labels

Every identifier and every label within a program text must have a unique defining point for an associated region which is part of the program text (see section 13.1).

- **Labels**

The defining point of a label is its occurrence in a label declaration part (see chapter 4) and the associated region is the total block closest-containing the defining point.

- **Constant identifiers, type identifiers, variable identifiers, procedure identifiers and function identifiers**

The defining point of the identifier is its occurrence in its definition or declaration (see chapters 5 to 8). A procedure or function identification does not contain a defining point, but rather an applied occurrence of the procedure-identifier or function-identifier (see sections 8.1, 8.2).

The associated region is the total block closest-containing the defining point. If the defining point is closest-contained in a package specification, the region is the entire package block. If, however, the defining point is closest-contained in a package body, the associated region is merely a package body. Thus, identifiers which are defined or declared in a package specification may be used directly in the package body, but not vice versa.

- **Package identifiers**

The defining point of a package-identifier is its occurrence in a package specification (immediately after PACKAGE). The occurrence of the package-identifier in a package body (immediately after PACKAGE BODY) is not a defining point, but rather an applied occurrence of the package-identifier.

The region for the defining point of a package-identifier is the entire program text, except for those compilation units which do not list the package-identifier in a WITH list of their context specification.
If the package-identifier is listed in the WITH list of the context specification of a package specification, the region of the package-identifier also includes the associated package body, regardless of whether the package-identifier is also named in a WITH list of the context specification of this package body.

- **Imported identifiers**

The defining point of an imported identifier is its occurrence in a USE list. The region for the defining point of an identifier imported into a package specification (i.e. one that is listed in a USE list of the context specification of the package specification) is the entire associated package block. The region for the defining point of an identifier imported into a main program is the program-block. Imported identifiers may therefore be used directly within their region (or more precisely, the scope; see section 12.3) without being preceded by the package-identifier.

- **Field identifiers**

The defining point of a field-identifier is its occurrence in a field designator. The associated region is the RECORD type closestcontaining the defining point (see section 6.3.3).

- **Constant identifiers of an enumerated type**

The region for the defining point of a constant-identifier which is defined as an enumerated-type value is the entire block closest-containing the enumerated type (see section 6.2.5). This means that, in the following example, the region for the defining point of "red" is the entire surrounding block, and not merely the RECORD type:

```
RECORD
   color : (red, yellow, green);
   ...
END
```

If the enumerated type is closest-contained in a package specification, the region for the constant-identifier in the enumerated type is the entire package block; if the enumerated type is closest-contained in a package body, the region is the package body.

- **Parameter identifiers**

The defining point of a parameter-identifier is its occurrence in a formal parameter list. The associated region is the formal parameter list closest-containing this defining point. If the formal parameter list is part of a procedure or function declaration for which there is a procedure- or function-block, the defining point of a parameter identifier is at the same time the defining point for a variable-identifier, procedure-identifier or function-identifier (depending on the type of parameter involved) for the region that constitutes the procedure-block or function-block.

- **Bound identifiers**

The defining point of a bound-identifier is its occurrence in a conformant array schema. The associated region consists of the formal parameter list closest-containing the defining point, plus the associated procedure-block or function-block.

- **Field designator identifiers in a WITH statement**

The RECORD-variable in a WITH statement is the defining point of all field identifiers declared in the associated RECORD type as field designator identifiers. The associated region (in which these field designator identifiers may be used directly) is the statement in the WITH statement (see section 10.5).

- **Required identifiers**

Identifiers for required constants, types, procedures and functions have an imaginary defining point (in front of the program text) whose region is the entire program text.

This does not apply to the required identifiers Input and Output, which stand for Text-type variables.
In Pascal-XT, Input and Output have their defining point in the package specifications of the required package Input_Package or Output_Package
(see also section 11.5).

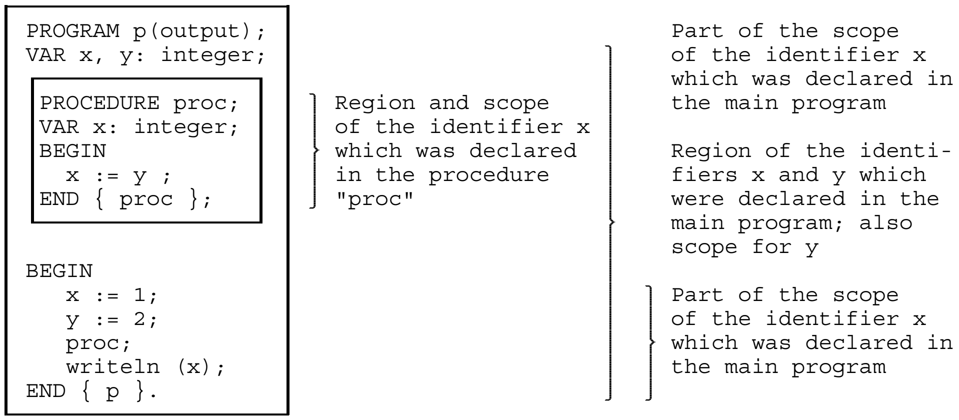
In Standard Pascal, Input and Output have an imaginary defining point in front of the main-program-block if and only if they occur in the program parameter list.

Cross-references

Label declaration:	4
Constant definition:	5
Type definition:	6
Variable declaration:	7.1
Procedure declaration:	8.1
Function declaration:	8.2
Formal parameters:	8.5
Bound identifiers:	8.5.4
WITH statement:	10.5
Program text:	13.1
Required identifiers:	A.2

Scopes and the Use of Identifiers

The use of an identifier (label) is only possible within the scope of its defining point. The scope of a defining point is its region, minus the regions of other defining points of like-named identifiers (labels) contained therein. If, in other words, as in Fig. 12-2, an identifier x in the main program has a defining point D1 and a like-named identifier has a defining point D2 in the procedure "proc", the scope of D1 is interrupted by the region of D2, so that only D2, but not D1, is valid within the procedure "proc".



```
PROGRAM p(output);
VAR x, y: integer;
  PROCEDURE proc;
  VAR x: integer;
  BEGIN
    x := y ;
  END { proc };

BEGIN
   x := 1;
   y := 2;
   proc;
   writeln (x);
END { p }.
```

Region and scope of the identifier x which was declared in the procedure "proc"

Part of the scope of the identifier x which was declared in the main program

Region of the identifiers x and y which were declared in the main program; also scope for y

Part of the scope of the identifier x which was declared in the main program

Fig. 12-2 Example of regions and scopes

In the region of a defining point of an identifier (label), there must be no like-named identifiers (labels) having a defining point for the same region. Thus, different but like-named identifiers (labels) cannot be declared or defined in the same block. The following type definition part is therefore illegal, since there are two different defining points in the same block for the like-named identifiers "unknown":

```
TYPE
   color = (red, yellow, green, unknown);
   form  = (round, square, unknown);
             { illegal due to double definition of "unknown" }
```

Identifiers (labels) may only be used within their scope. Their use then establishes a relation to the sole defining point of a like-named identifier (label) in this scope. For this reason, the use of the identifier x in the statement part of procedure "proc" (Fig. 12-2) refers to the defining point D2 of x; the variable x declared in the main program is left unchanged by the statement x := y. Thus, the program output is 1. The use of y within the procedure, on the other hand, refers to the variable y which is declared in the main program.

The defining point of an identifier (label) must precede all associated uses of that identifier (label) in the program text, except in two cases:

– New Pointer types must use a type-identifier as a domain type in front of their defining point, in so far as the applied occurrence and the defining point are closest-contained in the same declaration part (see section 6.4).

– Program parameters are applied occurrences of identifiers which are only declared later in the main-program-block or package block.

On the basis of these rules, the program below is errored:

```
PROGRAM errored;

   CONST max = 13;         { defining point D1 for max            }

   PROCEDURE p;            { start of region for D2               }

      TYPE t = 1..max;     { applied occurrence of defining point }

      VAR max: t;          { defining point D2 for max            }

   BEGIN
   END;                    { end of region for D2                 }

BEGIN
END.
```

In the main program, there is a defining point D1 for an identifier "max" with the entire main-program-block as its region. In the variable declaration within the procedure p, there is a defining point D2 for a second identifier "max" whose region is the entire procedure-block, i.e. the range of the procedure-block lying in front of the declaration of "max". It follows that this entire procedure-block is excluded from the scope of D1. The applied occurrence of the identifier "max" in the type definition within procedure p can therefore only refer to the defining point D2, since this applied occurrence does not lie in the scope of D1. In this case, however, this applied occurrence of "max" precedes the associated defining point D2, which is illegal.

Since the package-identifier in a package body constitutes the applied occurrence of an identifier, the associated defining point of this package-identifier, and thus the associated package specification in the program text (the imagined succession of all compilation units; see chapter 13), must precede the package body. Similarly, in the program text, the package specification of a package named in a WITH list of a context specification must precede the compilation unit starting with this context specification.

If a package-identifier is listed in a WITH list of a context specification of a compilation unit, then all identifiers in this compilation unit which are directly defined or declared in the associated package specification may be used by prefixing the package-identifier:

```
constant-name   = [package-identifier"."] constant-identifier.

type-name       = [package-identifier"."] type-identifier.

variable-name   = [package-identifier"."] variable-identifier.

procedure-name  = [package-identifier"."] procedure-identifier.

function-name   = [package-identifier"."] function-identifier.
```

In these constructs, the scope of the identifier following the period "." is formally excluded from all regions defined in section 12.2. Instead, the package-identifier is at the same time a new defining point for all identifiers which already have a defining point for the region of the associated package specifications. The region of these new defining points is then identical to the scope of the identifier following the ".", which is excluded from all other regions.

The same applies by analogy to selected objects (see section 9.6.3).

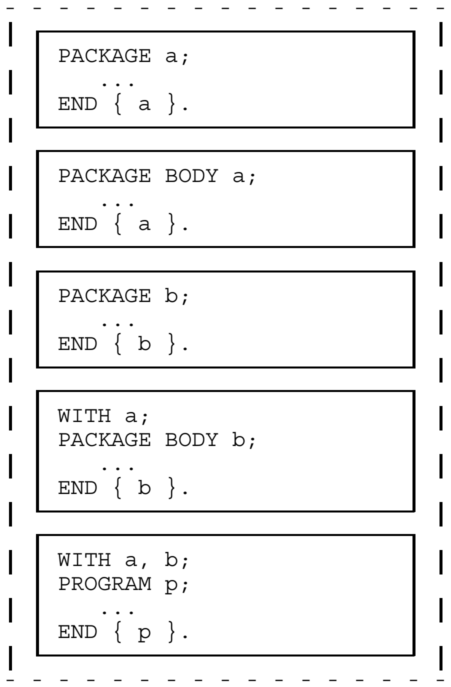
Structure, Compilation and Execution of Programs

This chapter describes the structure of an executable program, the characteristics of separate compilation, and program execution.

Program Structure

In Pascal-XT, a program consists of exactly one main program and any number of packages. A program can be considered as a succession of packages and the main program. This imagined succession of all compilation units belonging to a program is called a program text.

In Standard Pascal, a program consists solely of the main program.



```
PACKAGE a;
    ...
END { a }.

PACKAGE BODY a;
    ...
END { a }.

PACKAGE b;
    ...
END { b }.

WITH a;
PACKAGE BODY b;
    ...
END { b }.

WITH a, b;
PROGRAM p;
    ...
END { p }.
```

Fig. 13-1 Example of a program text

Note

There may be more than one different package body for a single package.
A program, however, is only allowed to contain one body of a package.

Example

In this example, the package "stack" from section 11.2 is used as a main program.

```
WITH stack;
PROGRAM calc (Input);
VAR
   i, j: Integer;
BEGIN
  FOR i := 1 to 6 DO BEGIN
      Read (Input, j);
      stack.push (j);
      END;
   WHILE NOT stack.is_empty DO BEGIN
      stack.pop (j);
      { process j }
      END;
END { calc }.
```

Compilation Units and Compilation Sequence

Package specifications, package bodies and main programs are all compilation units.

```
compilation-unit = package-specification
                 | package-body
                 | main program .
```

Each compilation unit is stored in its own source file, and is compiled separately from all other compilation units.

The rules governing the sequence in which compilation units are compiled can be derived directly from the scope rules and the relations specified in the context specifications:

– A compilation unit must be compiled in accordance with all package specifications listed in its WITH list.

– A package body must be compiled in accordance with the associated package specification.

A compilation unit is dependent on a package specification whenever the latter is listed in the WITH list of the compilation unit. Any change made to the package specification (e.g. modification of constants, identifiers, operators, WITH lists, etc.) invalidates all of the dependent compilation units. Therefore, once a modified package specification has been compiled, all dependent compilation units must be recompiled. A change made to a package body will not affect other compilation units.

Note

The manner in which compilation units are stored and the compiler accesses the package specifications listed in the WITH clauses is described in section 21.2.

Example

```
PACKAGE a;
  ...
END.


WITH a;
PACKAGE b;
  ...
END.


WITH b;
PACKAGE c;
  ...
END.
PACKAGE d;
  ...
END.


WITH c, d;
PROGRAM example;
  ...
BEGIN
END.
```

A change made to package specification b makes it necessary to recompile the dependent package specification c and, as a result, the main program "example" as well. Because package specifications b and c are recompiled, the associated package bodies must obviously be recompiled, too. Package specifications a and c are not affected by these changes.

Cross-references

Main program:	11.1
Package specification:	11.2.1
Package body:	11.2.2
WITH list:	11.3.1
Scope rules:	12.2

Executing a Program or Subprogram

Executing a Block

To execute a block, a separate set (incarnation) of the variables declared in the block is set up. The variables stored in this incarnation are different from the variables of any other incarnation of the same or a different block. In particular, each recursive incarnation of a procedure or function has its own set of variables. The lifetime of the variables stored in an incarnation ends when block execution is terminated (except in the case of package blocks; see section 13.3.3). Execution of a block consists in the execution of the statements in its statement part, and is terminated when

- the final statement in the statement part of the block has been processed;
- a RETURN statement is executed;
- a GOTO statement is executed whose jump label is declared and defined outside the block;
- an exception situation occurs (e.g. by means of Raise) for which there is no exception handler in the block.

In contrast, identified variables live from the moment they are created by a New call until the end of main program execution, or until identifying values referring to them are destroyed by a Dispose or Release call.

Cross-references

Identified variables:	7.2, 9.6.4, 18.3
GOTO statement:	10.1.4
RETURN statement:	10.1.6
Block:	12.1
Raise:	14, 15.11
New, Dispose, Release:	15.2, 18.3

Executing a Subprogram

Execution of a procedure-block or a function-block is activated by a procedure call (also known as a procedure statement) or by a function call (also known as a function designator).

Cross-references

Procedure call:	8.7
Function call:	8.7, 9.6.1

Executing a Program

As with subprogram block execution, variables declared directly in a main-program-block or a package block have, as their lifetime, the entire duration of program execution.

Program execution starts with the initialization of all packages belonging to the program, followed by execution of the main-program-block.

A package is initialized by executing the package block. Each package is initialized exactly once. If, while the statement part of a package block is being executed, variables declared in a foreign package specification are to be accessed or procedures and functions declared in a foreign package specification are to be called, then the foreign package must be initialized beforehand. This is done in an implementation-dependent sequence. To force a package to be initialized at the right time, the required procedure Elaborate (see section 15.12) can be called.

Possible runtime errors

`Memory_Error`	- Program execution cannot continue due to a lack of memory space (e.g. when a subprogram is called or in the case of New).
`Elab_Error`	- Initialization of the packages of a program cannot continue since loops were generated during initialization through the use of the required procedure Elaborate.
`unpredictable effects`	- The identifiers of the program parameters of the main program and all associated packages do not differ pair-by-pair, except for Input and Output. - The names of all packages belonging to a program and the name of the main program do not differ pair-by-pair.

Cross-references

Main program block:	11.1
Package block:	11.2
Package specification:	11.2
Block:	12.1
Elaborate:	15.12

Exception Handling

This chapter describes those extensions to the standard in Pascal-XT which are used for handling errors and exception situations that can occur when a program is running.

An exception is an event which interrupts normal program execution.
It may occur (be triggered) implicitly while a statement is being executed, or explicitly by calling the required procedure Raise. The reaction to an exception is referred to as exception handling. Actions to be performed may be specified in the EXCEPTION part of a compound statement. If an exception occurs, program execution continues in the EXCEPTION part (if present) of the compound statement.

The required procedure Raise (see section 15.11) can be employed by the user to create an exception situation or to propagate an exception which has already occurred. The exception number is passed as a parameter.

Once an exception situation has occurred, the required function Error_Number can be used to query the most recent exception.

Cross-references

Compound statement:	10.2
Raise:	15.11
Error_Number:	15.11

Predefined and User-defined Exceptions

An exception is represented by an Integer number. The negative numbers, including zero, are reserved for Pascal-XT implementations; the positive numbers may be employed at the user's discretion.

The reserved exceptions with the numbers -1 to -16 are linked to required identifiers as constants (see also section 5.2). These exceptions occur in the following situations (for details see Appendix A.5):

Numeric_Error	= -2	Occurs when there is an overflow during the computation of an arithmetic expression or a required arithmetic function, or when the real number assigned to a Short_Real variable is too large.
Range_Error	= -3	Occurs when an ordinal value does not lie in the permissible value range.
Set_Error	= -4	Occurs when not all members of the set lie in the value range of the relevant SET type.
String_Error	= -5	Occurs during character string processing when conditions regarding length are violated.
Index_Error	= -6	Occurs during object indexing when the value of the index expression does not lie in the permissible value range.
Pointer_Error	= -7	Occurs when a Pointer object is dereferenced with the Pointer value NIL.
Variant_Error	= -8	Occurs when a component in an inactive variant of a RECORD type is accessed. This error only occurs if the variant part containing this variant has a tag field.
Case_Error	= -9	Occurs in a CASE statement when there is no CASE constant for the value in the case index and no ELSE part is specified.
File_Error	= -10	Occurs during file access when the file is in the wrong mode or there are implementation-dependent restrictions (e.g. internal buffer is too small).
Eof_Error	= -11	Occurs during an attempt to read beyond the end of a file.
Open_Error	= -12	Occurs when a file cannot be opened (e.g. file nonexistent, access-protected, etc.).
Read_Error	= -13	Occurs during reading (Read, Readstring) of an Integer

or Real number when the number is syntactically errored
or its value is too large.

Memory_Error	= -14	Occurs during program execution when there is not enough main memory (e.g. when calling subprograms or the required procedure New).
Break_Error	= -15	Occurs when program execution is interrupted by the user.
Elab_Error	= -16	Occurs when initialization of the packages in a program cannot be continued because the use of the required procedure Elaborate creates loops during initialization.
System_Error	= -1	Occurs as a blanket term for other errors.

The detection of these predefined exceptions is defined by the Pascal-XT implementation, possibly in conjunction with the Check option.

Notes

– If the Check option is deactivated (see chapter 16) the errors Numeric_Error, Range_Error, Set_Error, String_Error, Index_Error, Pointer_Error, Variant_Error and Case_Error are generally not detected and cause unpredictable effects (see section 2.3). As an implementation-dependent feature, however, a Pascal-XT implementation can also detect errors when the Check option is deactivated.

– A Pointer_Error frequently occurs as a consequent error if, for example, program errors cause memory to be overwritten, or *Pointer*-objects are dereferenced whose values are undefined.

– All user-defined exceptions in a program should differ pair-by-pair. Only then can the origin of the exception be determined unambiguously in an EXCEPTION part.

Cross-references

Errors:	2.3
Predefined exceptions:	5.2, A.5
Assignment-compatibility:	6.6.3
Program execution:	13.3
Format denoter:	15.1
Check option:	16.2

EXCEPTION Part

The reaction to one or more exceptions can be defined in an exception handling part (EXCEPTION part). This part is an extension of the compound statement. It consists of a sequence of statements located after the keyword EXCEPTION. The syntax for the extended compound statement is thus as follows:

```
compound-statement = "BEGIN"
                      statement-sequence
                      [ EXCEPTION-part ]
                     "END".

EXCEPTION-part     = "EXCEPTION" statement-sequence.
```

Thus, an EXCEPTION part may be specified in the statement part (compound statement) of a subprogram, a main program or a package body, or in any other compound statement.

If no exception occurs when a statement sequence in a compound statement with EXCEPTION part is processed, the statements in the EXCEPTION part are not executed.

If, however, an exception does occur when the statement sequence in a compound statement with EXCEPTION part is processed, all remaining statements up to the keyword EXCEPTION are skipped, and execution resumes with the first statement in the EXCEPTION part. Once the statement sequence in the EXCEPTION part has been executed normally, the exception is considered to be handled, and processing of the compound statement terminated normally. If an exception situation again occurs when the statements in the EXCEPTION part are executed, of if another exception is created by means of Raise, the exception is said to be propagated (see section 14.3).

Example 14-1

```
   VAR
      source : Text;
      s      : String;

   BEGIN
      Reset (source);                                  {1}
{2}{3}
      WHILE NOT Eof (source) DO BEGIN                  {1}     {3}
         Read    (source, s);                          {1}     {3}
         Writeln (Output, s);                          {1}
         Readln  (source);                             {1}
         END;                                          {1}
   EXCEPTION
      IF Error_Number = Open_Error THEN                    {2}{3}
         Writeln (Output, 'Error when opening the file!')   {2}
      ELSE IF Error_Number = String_Error THEN                 {3}
         Writeln (Output, 'Input line is too long!')           {3}
      ELSE
         Raise (0);
   END;
```

In this example, comment braces indicate the lines which are to be executed when certain situations occur:

{1} Statements to be executed under normal conditions.

{2} Statements to be executed in case of error with Reset (Open_Error). The exception is considered to be handled.

{3} Statements to be executed when input line is too long (String_Error). The exception is considered to be handled and the read operation is terminated.

All other exceptions are propagated by Raise (0) since they cannot be handled.

Note

If an exception cannot be handled in an EXCEPTION part, the same exception should be propagated with Raise (0) since otherwise an unhandled exception situation will be incorrectly considered handled.

Cross-references

Procedure block:	8.1
Function block:	8.2
Compound statement:	10.2
Main program block:	11.1
Package body:	11.2
Package block:	11.2

Block: 12.1

How to Handle Exceptions

The way to handle an exception occurring when the statement sequence in a compound statement is processed depends on whether the compound statement contains an EXCEPTION part. If it does, then the statement sequence contained in this EXCEPTION part will be executed.

If the compound statement does not contain an EXCEPTION part, or if an exception again occurs in the EXCEPTION part, then the following cases may occur:

1) An exception situation in a compound statement which does not represent the entire statement part of a block, is again generated immediately behind the keyword END in this compound statement (see example 14-2). In the case of nested compound statements, the relevant EXCEPTION part can be determined on the basis of the static formulation of the program.

2) An exception situation in the compound statement (statement part) of a procedure-block or function-block is again generated at the call point of the subprogram (see example 14-3). Thus, the relevant EXCEPTION part cannot be determined from the static formulation of the program, but must be established dynamically by retracing the subprogram calls.

3) An exception situation in the compound statement (statement part) of a main-program-block or package block causes the program to abort. The error is handled by the relevant operating or runtime system.

An exception is propagated by being generated again with the Raise (Error_Number) or Raise (0) call. Raise (Error_Number) generates the same exception again and thereby changes the location at which the exception originated. Raise (0), on the other hand, propagates the same exception without changing the location at which the exception originated (see section 15.11 and examples 14-4 and 14-5).

Example 14-2

```
VAR
   i : Long_Integer;

BEGIN
   FOR i := 1 TO 5 DO BEGIN
      Reset (file);
      EXIT;
   EXCEPTION
      IF (i < 5) AND (Error_Number = Open_Error) THEN
         wait
      ELSE
         Raise (Error_Number)
   END;                                 {1}

   { processing the file }
   ...
END                                     {1}
```

If Raise is called in the EXCEPTION part, then, in accordance with Rule 1, the exception is generated again at the locations marked {1}.

Example 14-3

```
PROCEDURE r;
BEGIN
   ...
END {r};

PROCEDURE q;
BEGIN
   r;
EXCEPTION
   ...              { exception handling A2 }
END {q};

PROCEDURE p;
BEGIN
   q;
EXCEPTION
   ...              { exception handling A1 }
END {p};
```

The following situations may occur:

1) If an exception occurs in the statement part of procedure p, then EXCEPTION part A1 is responsible for handling it.
2) If an exception occurs in the statement part of procedure q which is called by p, then EXCEPTION part A2 is responsible. Once A1 has terminated normally, control returns to the call point of q (in the statement part of p). If an exception again occurs in A2, the q is aborted and the exception is again created at the call point of q (in p). There, the EXCEPTION part A1 is now responsible for handling it.
3) If an error occurs in procedure r which is called by q, then r is aborted and the exception is created again at the call point in q. The exception is handled in A2.

Note

When an exception occurs, note that actions may have to be performed in the EXCEPTION part to ensure the consistency of the data. In particular, before the EXCEPTION part is abandoned, you must ensure that the function-identifier has been assigned a value in the compound statement of a function.

Example 14-4

Error propagation with Raise (Error_Number). Here the original location of the error is lost.

```
PROGRAM quadr_equation (Input, Output);

PROCEDURE get_value (name: String; VAR value: Real);
BEGIN
   Writeln ('Please enter value for ', name, ':');
   Read (value); ____________________________________________ (1)
END;

PROCEDURE solve;
VAR
  p, q, d: Real;
BEGIN
   Writeln;
   Writeln ('Quadratic equation x**2 + p*x + q = 0');
   get_value ('p', p);
   get_value ('q', q);
   d := Sqrt (Sqr (p) / 4 - q);
   IF d = 0
      THEN Writeln ('x = ', -p / 2)
      ELSE Writeln ('x = ', -p / 2 + d, ' or ', -p / 2 - d);
EXCEPTION
   IF Error_Number = Numeric_Error
      THEN Writeln ('The equation has no solution')
      ELSE Raise (Error_Number); _____________________________ (2)
END;

BEGIN (*PROGRAM quadr_equation*)
   WHILE True
      DO solve;
EXCEPTION
   IF Error_Number = Eof_Error
      THEN Writeln ('Goodbye')
      ELSE Raise (Error_Number); _____________________________ (3)
END.
```

(1) If an error occurs while the number is being read, a Read_Error is triggered. The location where this error originated thus lies in the procedure get_value (1).

(2) The Read_Error is propagated, i.e. triggered again. This causes the original location of the error (1) to be lost. The new error location is (2).

(3) Same behavior as (2). The new error location is now the main program (3).

This type of error propagation hampers error diagnostics since the error location is lost. The response of the program is described in detail in the User's Guide.

Example 14-5

With Raise (0) the same exception is propagated. Here, the original error location is retained for purposes of error diagnostics. Raise (0) does not trigger an exception with the number 0.

```
PROGRAM quadr_equation (Input, Output);

PROCEDURE get_value (name: String; VAR value: Real);
BEGIN
   Writeln ('Please enter value for ', name, ':');
   Read (value); ———————————————————————————————————————————— (1)
END;

PROCEDURE solve;
VAR
  p, q, d: Real;
BEGIN
   Writeln;
   Writeln ('Quadratic equation x**2 + p*x + q = 0');
   get_value ('p', p);
   get_value ('q', q);
   d := Sqrt (Sqr (p) / 4 - q);
   IF d = 0
      THEN Writeln ('x = ', -p / 2)
      ELSE Writeln ('x = ', -p / 2 + d, ' or ', -p / 2 - d);
EXCEPTION
   IF Error_Number = Numeric_Error
      THEN Writeln ('The equation has no solution')
      ELSE Raise (0); —————————————————————————————————————— (2)
END;

BEGIN (*PROGRAM quadr_equation*)
   WHILE True
      DO solve;
EXCEPTION
   IF Error_Number = Eof_Error
      THEN Writeln ('Goodbye')
      ELSE BEGIN
         Writeln ('Program aborted');
         Raise (0); ——————————————————————————————————————— (3)
         END;
END.
```

(1) A Read_Error occurred while reading the number.

(2) The exception is propagated by Raise (0). Here, the original error location (1) is retained in the get_value procedure.

(3) Same behavior as (2). The original error location (1) is retained.

This type of error propagation considerably simplifies error diagnostics as compared to example 14-4. The response of the program is described in detail in the User's Guide.

Cross-references

Procedure block:	8.1
Function block:	8.2
Compound statement:	10.2
Main program block:	11.1
Package body:	11.2
Package block:	11.2
Block:	12.1
Error_Number	15.11
Raise	15.11

Exception Handling and Optimization

This section describes the conditions under which an implementation may execute statements and expressions in a sequence other than the one prescribed by the Pascal language.

Insofar as Pascal-XT specifies rules for the sequence in which certain actions are to be performed (canonical rules), an implementation is allowed to choose an alternative sequence only if this does not affect the program. In particular, the alternative sequence must not give rise to any exception situations that would not have occurred anyway if execution had followed canonical sequence.

If, however, an exception situation does occur when working in canonical sequence, a program must not assume that the expressions in a compound statement will be evaluated in canonical sequence. Since optimization causes expressions to be executed in a non-canonical sequence, it may happen during this preliminary evaluation that a different exception occurs than would have happened without optimization. Nevertheless, we can guarantee that no additional exception situations will arise as a result of optimization.

Example

```
VAR
   i,n : Integer;

BEGIN
   n := 0;
   FOR i := 1 TO 5 DO
      n := n + i ** f[a];  { let f and a be global }
EXCEPTION
   IF Numeric_Error THEN
      Writeln (Output, 'Error during computation')
   ELSE
      Writeln (Output, 'unknown error')
END.
```

Let f and a be global variables. The evaluation of f[a] is independent of the FOR loop and may therefore be performed prior to the loop and even prior to the statement "n := 0", even if it leads to an exception situation. Accordingly, the value of n may be undefined within the EXCEPTION part. The evaluation of f[a] may, however, not take place prior to BEGIN, since otherwise any resultant exception would be handled by a different EXCEPTION part.

Required Subprograms

This chapter describes all of the required subprograms (procedures and functions), subdivided into groups related by topic. Within each group, the subprograms are listed in alphabetical order. An alphabetical list of all required subprograms can be found in Appendix A.4.

The identifiers of the required subprograms are not word symbols (keywords), and may therefore be redefined if desired.

Required subprograms may be called in the same way as user-defined subprograms:

However, standard procedures/functions may not be passed as procedural/functional parameters.

Moreover, these subprograms do not necessarily obey all of the rules set down for user-defined subprograms (see chapter 8).

File Processing Subprograms

Assignfile (f, ext)

This procedure assigns, to a FILE variable f, a file existing outside the program.

"f" is a variable of any FILE type.
"ext" is a character string expression and contains the implementation-defined description of the physical file assigned to the FILE variable (see section 22.2).

If necessary, Assignfile implicitly closes the file previously assigned to FILE variable f.

Implementation-defined characteristic

The way the physical file is described in the required procedure Assignfile and the effect of this procedure are implementation-defined.

Possible runtime errors:

`File_Error`	`- With Assignfile (f, ext), the description of the physical file in the operand "ext" is errored.`

Note:

In Standard Pascal, only files specified in the program parameter list may be assigned physical files. By using Assignfile, it becomes possible to do this with any FILE variables, even for local FILE variables and those created dynamically with New.

Cross-references

FILE type:	6.3.5
Variables:	7.2
Input/Output:	16

Eof (f)
Eof

The Eof (f) function returns the Boolean value True when the end of file f is reached (EOF); otherwise, it returns the Boolean value False.

If the parameter f is not specified, the function is applied to the required textfile Input.

If the file was opened for writing, the function Eof always returns the value True.

Following Reset, Eof is True when the file is empty. Following Get, Eof is True if, prior to Get, the final component of the file was in the buffer variable. In all other cases Eof is False. If Eof is True, calling Get, Read, Readln and Eoln will cause the runtime error Eof_Error.

If Eof is True, the buffer variable f↑ is undefined.

Possible runtime errors:

`File_Error`	`- File f is undefined prior to the Eof (f) call (see section 7.3).`

Cross-references

Buffer variables:	8.2, 16.1
Get:	15.1
Defined variables:	7.3

Eoln (f)
Eoln

Eoln is a function for recognizing the end-of-line component in text files. Eoln can only be used on textfiles which have been opened for reading. If the parameter f is omitted, the function is applied to the required textfile Input.

The Eoln function returns the value True if the last item to be read was the end-of-line component. The buffer variable then contains a blank (' '). Otherwise, Eoln returns the value False.

Possible runtime errors:

`File_Error`	`- File f is undefined prior to the Eoln (f) call.`
`Eof_Error`	`- When Eoln (f) is called, the end of file marker is already reached, i.e. Eof (f) is True.`

Cross-references

Textfiles:	6.3.5.2, 16
End-of-file component:	16.1

Get (f)

Get (f) reads the next component from file f and includes it in the buffer variable f↑. f may be any FILE-type variable which is open for reading. Eof (f) must be False before Get (f) is called.

If a component was read, then Eof (f) has the value False. If it was not possible to read another component (end of file), then the buffer variable f↑ is undefined and Eof (f) has the value True.

Possible runtime errors:

`File_Error`	- Prior to the Get (f) call, file f was not opened for reading. - File f was undefined prior to the Get (f) call.
`Eof_Error`	- Prior to the Get (f) call, Eof (f) was True.

Example

The components of file f were read in sequentially and processed. With Reset (f) the first component is read to the buffer variable.

```
VAR
   f : FILE OF component_type;
BEGIN
   Reset (f);
   WHILE NOT Eof (f) DO BEGIN
      process (f↑);
      Get (f);
      END;
END;
```

Page (f)
Page

The Page procedure is used for formatting the pages of textfiles. The text which is output following Page (f) is put onto a new page, provided the output device possesses a page control option. The way in which form feed takes place is implementation-defined (see User's Guide [1,2]). If an output line has not been terminated with Writeln (f) before Page is called, Page will perform this Writeln (f) implicitly.

Page can only be applied to a textfile which is open for writing. The procedure is applied to the required textfile Output if no FILE variable was specified.

Implementation-defined characteristic

The effect of the required procedure Page on textfiles is implementation-defined.

Implementation-dependent characteristic

The effect of reading a textfile which was being generated when the required procedure Page was applied is implementation-dependent. However, this effect can be defined for a particular implementation.

Possible runtime errors:

`File_Error`	`- File f was not opened for writing prior to the Page call.`
	
	`- File f is undefined prior to the Page (f) call.`

Put (f)

The procedure Put (f) may be applied to a FILE variable which is open for writing. The actual value of the buffer variable f↑ is appended to the end of the processed file. Following the Put call, the buffer variable f↑ is undefined and Eof remains True.

The Put procedure also offers the opportunity of writing components in abbreviated form. This form of the call is explained further below.

Possible runtime errors:

`File_Error`	- File f was not opened for writing prior to the Put call. - File f is undefined prior to the Put (f) call.
`unpredictable effects`	- The value of the buffer variable f↑ is undefined. - When an ARRAY is output in abbreviated form with Put (f, e) or Put (f, c1, ..., cn, e), the value of index expression e does not lie in the value range of the index type of the ARRAY. - When a variable string is output in abbreviated form with Put (f, e) or Put (f, c1, ..., cn, e), the value of index expression e is less than 1 or greater than the actual length of the string.

Example

File f is written sequentially with previously computed values.

```
VAR
   f : FILE OF component_type;
BEGIN
   Rewrite (f);
   WHILE NOT finished DO BEGIN
      f↑ := new_value;
      Put (f);
      END;
END;
```

Put (f, e)

The base type of FILE type f must be an ARRAY type, a variable string type, or a RECORD type with an ARRAY type or variable string type for the final field in its field list.

With an ARRAY type, the expression e must be assignment-compatible with the index type of the ARRAY type.
With a variable string type, the expression e must have an Integer type, and its value must not be less than 1 or greater than the actual value of the variable string.

With a Put call of this sort, an implementation is allowed to write the components to the file in abbreviated form.
With ARRAY [m..n], only components m to e are written to the file. The actual length of a character string must be ≤ e.

Example

```
VAR
   h : FILE OF ARRAY [1..100] OF Real;
   a : Integer;
   s : FILE OF String;

BEGIN
   ...

   Put (h, 45); { writes at least the first 45 real values }

   a := 23;
   Put (h, a);  { writes at least the first 23 real values }

   Put (s, a);  { writes s↑ if length (s↑) ≤ 23 }
END;
```

Put (f, c1, ..., cn)

The base type of FILE type f must be a RECORD type. It must contain nested variants to which the CASE constants c1, ..., cn belong. These CASE constants must be enumerated in the sequence defined by the nesting of the variant parts. These variants must be initialized to the value of f↑.
Variants which are not listed must lie in a deeper nesting level then cn.
With a Put call of this sort, an implementation may write an abbreviated record to the file, i.e. one that only has space for the RECORD fields of these variants.

Example

Given the following RECORD type, consisting of nested variants which are referred to in the examples below:

```
TYPE
  char_range = ’0’..’1’;
  t = RECORD
         t1: Integer;
         CASE t2: Boolean OF
            False:
               (t21: Real;
                CASE t22: char_range OF
                   ’0’: (t221: ARRAY [0..10] OF Integer);
                   ’1’: (t222: Short_Integer));
            True: (t23: Char);
      END;
VAR
  g: FILE OF t;
```

The following Put calls are permitted. The comments indicate how the length is calculated. Writing takes place in abbreviated form only if provisions have been made for it in the relevant implementation.

```
Put (g, False);      { writes t1, t2, t21, t22, (t221 or t222) }

Put (g, False, ’0’); { writes t1, t2, t21, t22, t221 }

Put (g, False, ’1’); { writes t1, t2, t21, t22, t222 }

Put (g, True);       { writes t1, t2, t23 }
```

Put (f, c1, ..., cn, e)

The base type of FILE type f must be a RECORD type with nested variants to which the CASE constants c1, ..., cn belong. These CASE constants must be enumerated in the sequence defined by the variant parts. The field list of the variant belonging to the final CASE constant cn must not contain any further nested variants, and the final field in this field list must have an ARRAY or variable string type.

With an ARRAY type, the expression e must be assignment-compatible with the index type of the ARRAY type.
With a variable string type, the expression e must have an Integer type, and its value must not be less than 1 or greater than the actual length of the variable string.

With a Put call of this sort, an implementation may write an abbreviated record to the file, i.e. one containing only the RECORD fields of the variants determined by c1 .... cn. If the final RECORD field is an

ARRAY [m..n], only components m to e are written to the file. If the final RECORD field is a variable string, its actual length must be ≤ e.

Example

```
Put (g, False, '0', 2); { writes t1, t2, t21, t22,   }
                        {  t221[0], t221[1], t221[2] }
```

Read (f, v1, ..., vn)

The Read procedure is used for reading in values from a file f to the variables v1 to vn.

The file variable f may be of any FILE type except for the required FILE type Any_File. File f must be open for reading, and Eof (f) must be False.

For the number of parameters vi, $n \geq 1$.

Read (f, v1, ..., vn)

is defined by

BEGIN Read (f, v1); ...; Read (f, vn); END

Although the parameters vi must be variable accesses, they are not variable parameters. They may therefore also be components of packed structures.

- **Read - from a non-textfile**

  Read (f, v)

  If FILE variable f does not have the required FILE type Text, then

  Read (f, v)

  is defined by:

```
BEGIN    v := f↑; Get (f); END
```

  v must be a variable access. The value of the buffer variable f↑ must be assignment-compatible with the type of v.

Possible runtime errors:

File_Error	- Prior to the Read (f, ...) call, file f was not opened for reading. - File f is undefined prior to the Read (f, ...) call.
Eof_Error	- Prior to the Read (f, ...) call, the end of file marker is already reached, i.e. Eof (f) is True.
Numeric_Error	- While reading from a non-textfile with Read(f,v), the value of the buffer variable f↑ of the type Long_Real does not lie in the value range of the type Short_Real of variable v.
Range_Error	- While reading from a non-textfile with Read (f,v), the value of the ordinal-type buffer variable f↑ does not lie in the value range of the ordinal type of variable v.
Set_Error	- While reading from a non-textfile with Read (f,v), the value of the SET-type buffer variable f↑ does not lie in the value range of the SET type of variable v.
String_Error	- While reading from a non-textfile with Read (f,v), the actual length of the character string of type String in the buffer variable f↑ is greater than the maximum length of the variable string type of variable v. - While reading from a non-textfile with Read (f,v), the actual length of the character string of type String in the buffer variable f↑ is not equal to the length of the fixed string type of variable v.

Example

The components of file f are read sequentially and processed:

```
VAR
   f : FILE OF component_type;
   v : component_type;
BEGIN
   Reset (f);
   WHILE NOT Eof (f) DO BEGIN
      Read (f, v);
      process (v);
      END;
END;
```

Cross references

General files:	6.3.5
Assignment-compatibility:	6.6.2
Variable parameters:	8.5.2
Variable accesses:	7

- **Read - from a textfile**

Read (f, v1, ..., vn)
Read (v1, ..., vn)

If FILE variable f has the required type Text, then each variable vi (i = 1...n) may have the following types:
– the type Char or a subrange of Char,
– an Integer type or a subrange of an integer type,
– a Real type,
– a variable string type,
– a type PACKED ARRAY [1..n] OF Char.

If the type of a variable vi is an Integer type or a Real type, the value represented by the character string is moved to the variable vi.

If FILE variable f was omitted, reading takes place from the required textfile Input.

Possible runtime errors:

`File_Error`	- Prior to the Read (f, ...) call, file f was not opened for reading. - Prior to the Read (f, ...) call, file f was undefined.
`Eof_Error`	- Eof (f) became True when Read (f, ...) was called or when leading blanks and end-of-line components were being skipped.
`Numeric_Error`	- The value of the input Real number can be represented internally, but it lies outside the value range of the read parameter v.
`Read_Error`	- While an Integer number or Real number is being read from a textfile (with Read (f, v)), the following happens: - the number is syntactically incorrect or - the value of the input number is too large, thus it cannot be represented internally. With Integer numbers the value lies outside the range Long_Minint .. Long_Maxint ; with Real numbers the value lies outside the range -Long_Maxreal .. Long_Maxreal.
`Range_Error`	- The value of an input Integer number can be represented internally, but it lies outside the value range of the read parameter v.
`String_Error`	- With Read (f,v), the maximum length of the String variable v is less than the length of the input character string.

- **Reading a character**

  Read (f, v), where v is a Char-type variable

  If v is a variable of type Char or a subrange of Char, then Read (f, v) is defined by

```
   BEGIN  v := f↑; Get (f)  END
```

  The variable v contains the space character (blank) ' ' if, prior to the Read (f, v) call, Eoln (f) is True.

- **Reading an Integer value**

  Read (f, v), where v is an Integer-type variable

  If v is a variable of an Integer type or a subrange thereof, a string of characters is read. Leading blanks and end-of-line characters are skipped. A sign may appear directly in front of the number. The read operation is terminated when an input character can no longer be part of an Integer number. This character is then located in f↑.

  The input character string must correspond to a signed Integer number in accordance with the following general form:

```
integer-number  =  (sign) unsigned-integer-number.
sign            =  "+" | "-".
```

  The input character string is converted to an Integer value and moved to variable v. The Integer value must be assignment-compatible with the type of v.

- **Reading a Real value**

  Read (f, v), where v is a Real-type variable

  If v is a variable of a Real type, a string of characters is read. Leading blanks and end-of-line characters are skipped. A sign may appear directly in front of the number. The read operation is terminated when an input character can no longer be part of an unsigned Integer number or a Real number. This character is then located in f↑.

  The input character string must correspond to a signed number in accordance with the following general form:

```
real-number   =  [sign] ( unsigned-real-number
                        | unsigned-integer-number ).
sign          =  "+" | "-".
```

  The input character string is converted to a Real value and moved to variable v. The Real value must be assignment-compatible with the type of v.

– **Reading a String-type variable**

Read (f, v), where v is a String-type variable

All characters are read to the end of the current line and moved to v.
After Read (f, v), Eoln (f) is True and f↑ contains a blank (' ').

Read (f, v) is equivalent to:

```
BEGIN
   v := '';
   WHILE NOT Eoln (f) DO BEGIN
      v := Concat (v, f↑);
      GET (f);
      END;
END;
```

– **Reading a fixed-string variable**

Read (f, v), where v is a fixed-string variable with length n

Character input is terminated when one of the following two conditions is satisfied:

1) The number of input characters is equal to the length n of the character string type of v.
2) An end-of-line character was reached, i.e. Eoln (f) is True.
   If the number of input characters is less then n, the remaining components of v are padded with blanks (' ').

Read (f, v) with v of type PACKED ARRAY [1..n] OF Char is equivalent to:

```
FOR i := 1 TO n DO
   IF Eoln (f) THEN
      v[i] := ' '
   ELSE
      Read (f, v[i]);
```

Example

This example illustrates the possible effects of Read with different parameters. File f contains the following lines (blanks are represented by '_' for better visibility):

```
1. line: #_Hello
2. line: A123____3.1415PI
3. line: ____
4. line: 0001_DM_Finished

VAR
   f    : Text;
   i, j : Integer;
   r    : Real;
   c    : Char;
   s    : String;
   a    : PACKED ARRAY [1..3] OF Char;
BEGIN
   Reset (f);

   Read (f, c);      { c contains '#' }

   Read (f, s);      { s contains '_Hello' }

   Read (f, s);      { s contains '' because Eoln (f) = True }

   Readln;           { advance to next line }

   Read (f, c);      { c contains 'A' }

   Read (f, i);      { i contains 123 }

   Read (f, r);      { r contains 3.1415 }

   Read (f, a);      { a contains 'PI_' }

   Read (f, j);      { line 3 is skipped, j contains 1 }

   Read (f, a);      { a contains '_DM' }

   Read (f, s);      { s contains '_finished' }
END
```

The statement sequence is equivalent to:

```
BEGIN
   Reset (f);
   Readln (f, c, s, s);
   Read (f, c, i, r, a, j, a, s);
END
```

Readln (f, v1, ..., vn)
Readln (v1, ..., vn)
Readln (f)
Readln

FILE variable f must be of type Text. If f is omitted, Readln is applied to the required file Input. The permissible parameters v1 to vn are described under Read for textfiles.

When Readln is called, control is positioned to the beginning of the next line in the file.

Readln (f, v1, ..., vn) is equivalent to

```
BEGIN
   Read (f, v1); ...; Read (f, vn);
   Readln (f);
END
```

and Readln (f) is equivalent to

```
BEGIN
   WHILE NOT Eoln (f) DO Get (f);
   Get (f);
END
```

The remaining characters in the current input line (if any) are skipped. The first character in the new line is moved to the buffer variable f↑. If the end-of-file character has been reached, the buffer variable is undefined following Readln, and Eof is True.

Possible runtime errors: same as for Read (see above).

Note

The special characteristics of Readln in conjunction with interactive terminal input are described in chapter 19.

Reset (f)

Reset (f) opens a file f for reading, and reads the first component to the buffer variable f↑. If the file is empty, Eof returns the value True and buffer variable f↑ is undefined.

Implementation-defined characteristic

According to Standard Pascal, the effect of the required procedure Reset on one of the required textfiles Input or Output is implementation-defined.
In Pascal-XT, this characteristic has been made identical for all implementations: an Open_Error occurs.

Possible runtime errors:

`Open_Error`	`- When Reset was called, the required textfile Input or Output was specified.`
	`- When Reset (f) was called, file f was undefined (see section 7.3).`
	`- With Reset (f), the external file assigned to f cannot be opened for reading.`

Rewrite (f)

Rewrite opens file f for writing. Following Rewrite (f), the file is empty and can be rewritten. Eof (f) is True. The contents of f↑ are undefined. Any contents in the old file are lost.

Implementation-defined characteristic

According to Standard Pascal, the effect of the required procedure Rewrite on one of the required textfiles Input or Output is implementation-defined.
In Pascal-XT, this property has been made identical for all implementations: an Open_Error occurs.

Possible runtime errors:

`Open_Error`	`- When Rewrite was called the required textfile Input or Output was specified.`
	
	`- With Rewrite (f), the external file assigned to f cannot be opened for writing.`

Write (f, a1, ..., an)

The Write procedure is used for writing the values a1 to an to a file f.

The FILE variable f may be of any FILE type except for the required FILE type Any_File. File f must have been opened for writing.

The call

```
Write (f, a1, ..., an)
```

```
is defined by:
```

```
BEGIN Write (f, a1); ...; Write (f, an); END
```

- **Write - to a non-textfile**

  Write (f, a1, ..., an)

  If FILE variable f is not of the required FILE type Text, then

  Write (f, a)

  is defined by:

  ```
  BEGIN    f↑ := a; Put (f) END
  ```

  The expression a must be assignment-compatible with the buffer variable f↑.

Possible runtime errors:

File_Error	- File f was not opened for writing prior to the Write call. - File f is undefined prior to the Write (f) call.
Numeric_Error	- While writing to a non-textfile with Write (f, a), the value of expression a of type Long_Real does not lie in the value range of the buffer variable f↑ of type Short_Real.
Range_Error	- While writing to a non-textfile with Write (f, a), the value of an ordinal-type expression a does not lie in the value range of the ordinal type of the buffer variable f↑.
Set_Error	- While writing to a non-textfile with Write (f, a), the value of a SET-type expression a does not lie in the value range of the SET type of the buffer variable f↑.
String_Error	- While writing to a non-textfile with Write (f, a), the actual length of the character string is greater than the maximum length of the variable string type of the buffer variable f↑. - While writing to a non-textfile with Write (f, a), the actual length of the character string a of a variable string type is not equal to the length of the buffer variable f↑ of a fixed string type.

Example

The computed data are written sequentially to file f.

```
VAR
   f : FILE OF component_type;
   v : component_type;
BEGIN
   Rewrite (f);
   WHILE NOT finished DO BEGIN
      v := new_value;
      Write (f, v);
      END;
END;
```

- **Write - to a textfile**

  Write (f, a1, ..., an)
  Write (a1, ..., an)

  If the FILE variable f is of the required type Text, then

  ```
  Write (f, a1, ..., an)
  ```

  is defined by

  ```
  BEGIN Write (f, a1); ..; Write (f, an) END
  ```

  and each parameter ai may take one of three forms:

  ```
  a
  a : n
  a : n : m.
  ```

  If the FILE variable f is omitted, the required textfile Output is assumed.

  The value of expression a which is written to file f may have one of the following types:

  – the type Char or a subrange of Char,
  – the type Boolean,
  – an Integer type or a subrange of an Integer type,
  – a Real type or
  – a variable or fixed string type.

  With Integer and Real types, the value of the number is converted to a character string prior to writing.

  The entries n and m are used to format the output values, where n stands for the total output length and m for the number of digits following the decimal point. n and m are Integer-type expressions whose values must be greater than or equal to 1. If the type of a is a variable string type, then the value of n may also be 0.

  The format a is equivalent to the format a:n, where a predefined value is used for n depending on the type of a involved. For Integer-type, Real-type and Boolean-type parameters, this value of n is implementationdefined.

  The format a : n : m may be used only if expression a is of a Real type. This selects fixed-point representation, with m indicating the number of positions following the decimal point (see further below).

Possible runtime errors:

`File_Error`	- File f was not opened for writing prior to the Write call. - File f is undefined prior to the Write (f) call.
`Range_Error`	- With Write (f,a:n), the total output length n < 1, or n < 0 if a has a variable string type. - With Write(f,a:n:m), the total output length n < 1 or the number of digits following the decimal point m < 1.

– **Outputting a character**

Write (f, a) or
Write (f, a : n), where a is a Char-type expression

If a is an expression of type Char or a subrange of Char, then

Write (f, a)

is defined by

```
BEGIN    f↑ := a; Put (f) END
```

The effect of Write (f, a) is identical to that of Write (k, a:1). The Write (f, a:n) call causes an additional (n-1) blanks to be prefixed to a.

Example

This example outputs a pyramid of asterisks, with the position of the first asterisk in each line being determined by a length parameter.

```
PROGRAM pyramid d (Output);
CONST
   limit = 5;
   star  = '*';
VAR
   i : 1..limit;
   k : 0..8;
BEGIN
   FOR i := 1 TO limit DO BEGIN
      Write (star : (limit - i + 1));
         { writes an asterisk with leading blanks }
      FOR k := 1 TO 2 * (i - 1) DO
         Write (star);
      Writeln;
      END
END.
```

Program printout:

```
1. line:          *
2. line:         ***
3. line:        *****
4. line:       *******
5. line:      *********
```

– **Outputting character strings**

Write (f, a) or Write (f, a : n), where a denotes a string-type expression

If a is a fixed string type with k components, then k is the default value for the total output length.

If expression a is a variable string type, then the actual length of a (Length (a)) is the default value for the total output length.

If a length is specified, output is right-justified, and leading blanks are added if the actual length of the character string to be output is less than the total output length n. If, on the other hand, the character string is longer than the total output length, only its first n elements are output.

In the case of variable string values, the length specification may also be zero. In this case, no output takes place.

– **Outputting Integer values**

Write (f, a) or
Write (f, a : n), where a is an Integer-type expression

The value of expression a is written in decimal form to file f:

– a series of blanks (adapted to output length),
– a minus sign (’-’) if the expression is negative, and
– the value of the expression (decimal number).

The number of characters to be output is determined by the expression n if the form Write (f, a:n) was selected. Otherwise, an implementationdefined number of characters is output (see user’s guide).

If the statement Write (f, a:n) defines an output field whose length is too short to represent the value of the expression in decimal form, the required length will be substituted for n. If n is greater than required to output the number, the number will be padded with blanks to the specified length.

Implementation-defined characteristic

The default output length for Integer-type values is implementation-defined.

Example

For better visibility blanks are represented by '_'.

```
Write statement          output                        actual
                                                    output length

Write ( 1325:8)          ____1325                         8
Write (-1235:8)          ___-1235                         8
Write ( 9876:2)          9876                             4
Write (-9876:2)          -9876                            5
```

– **Outputting Real values in floating-point notation**

Write (f, a) or
Write (f, a:n), where a is a Real-type expression

The value of expression a is written to file f in floating-point decimal form with scale factor, rounded to the desired number of significant positions:

- a series of blanks (adapted to the output length),
- a minus sign ('-') if the expression is negative,
- the value of the mantissa (positions in front of and behind the decimal point) and
- the scale factor (exponent).

If the total output length n is omitted, an implementation-defined output length is assumed (see User's Guide). If n too small for maximum-precision output, decimal positions will be rounded off. If n is also too small for output with only one digit following the decimal point, it will be replaced be the requisite length. If n is greater than necessary, zeros will be appended to the decimal digits. In any case, either a minus sign or a blank appears in front of the leading digit.

Implementation-defined characteristics

- The default output length for Real-type values is implementation-defined.
- The representation of the scale factor (’E’ or ’e’) is implementation-defined, as is the number of decimal digits used for the scale factor when Real values are output in floating-point notation.

Example

For better visibility blanks are represented by ’_’.

```
Write statement                    impl.-def.            actual
                                     output          output length

Write (8.12231E17:12)              _8.12231E+17          12
Write (-10.052173:14)              -1.0052173E+01        14
Write (5.0123E-6:11)               _5.0123E-6            11
Write (0.5:13)                     _5.000000E-01         13
Write (12.3:9)                     _1.23E+01              9
Write (2.3128143561231E11:20)      _2.3128143561231E+11   20
Write (3.14:1)                     _3.1E+00               8
```

- **Outputting Real values in fixed-point notation**

  Write (f, a : n : m), where a is a Real-type expression

  The value of expression a is written to file f in fixed-point decimal form with m positions following the decimal point, as follows:

  - a series of blanks (adapted to the output length),
  - a minus sign (’-’) if the expression is negative, otherwise a blank (’ ’), and
  - the value of the expression (positions in front of and behind the decimal point).

  n is the total output length. If the n specified is too small, the least possible value is assumed. The minimum total output length results from the number of positions in front of and behind the decimal point, plus one position for the decimal point and another position for the sign (’ ’ or ’-’).

  m determines the number of positions to be output following the decimal point. If m is greater than required for maximum-precision output, zeros will be appended to the decimal positions. If m is smaller than required, the smallest position to be output will be rounded off.

Example

For better visibility blanks are represented by '_'.

```
Write statement                       output               actual
                                                        output length

Write ( 12219.378:10:2)               __12219.38             10
Write ( 12219.378:10:3)               _12219.378             10
Write (-12219.378:10:3)               -12219.378             10
Write (-12219.378:6:1)                -12219.4                8
```

– **Outputting Boolean values**

Write (f, a) or
Write (f, a : n), where is a Boolean-type expression

Output corresponds to the following format:

– a series of blanks (padding to output length),
– the character string 'TRUE' or 'FALSE'.

Write (f, a : n) is handled as follows:
Output takes place as if a variable string of the specified length were output.

– n > output length : leading blanks, text right-justified
– n < output length : the first n characters are output

Implementation-defined characteristics

– The notation (upper case/lower case) of the individual letters of the Boolean values True and False, when output, is implementation-defined for each letter.

– The default output length for Boolean-type values is implementation-defined.

Example

For better visibility blanks are represented by '_'.

```
Call                      impl.-def.
                           output

Write (1=2:10)            _____FALSE
Write (1=1:3)             TRU
```

Writeln (f, a1, ..., an)
Writeln (a1, ..., an)
Writeln (f)
Writeln

The FILE variable f must be of type Text, and it must be opened for writing. If f is omitted, Writeln is applied to the required textfile Output. The permissible parameters a1 to an are described under Write.

A writeln call causes an end-of-line component to be written to file f and control to be positioned to the start of a new line.

Writeln (f, p1, ..., pn) is equivalent to

BEGIN Write (f, p1, ..., pn); Writeln (f) END

and Writeln (f) is equivalent to

```
BEGIN
   f↑ := "end-of-line component";
   Put(f);
END
```

Possible runtime errors: same as for Write to textfiles.

Note

With buffered output to a data display terminal, not until Writeln has been entered is a line terminated and displayed on the screen.

Heap Management Subprograms

Dispose (p)

The parameter p must be an expression of a Pointer type other than the generic pointer type. Its value must reference an identified variable p↑ which was created with New. Dispose (p) releases the memory space required for the identified variable p↑.

By means of New, the Pascal-XT system can reuse the released memory space of p↑ to meet later requirements.

Afterwards, the identified variable p↑ can no longer be accessed, i.e. all identifying values pointing to it are rendered invalid by Dispose (p).

If the form New (q, ...) with multiple parameters was chosen (abbreviated initialization of an identified variable), the same form must also be chosen for Dispose.

Possible runtime errors:

`Pointer_Error`	`- When Dispose (p) is called, p has the value NIL.`

`unpredictable effects`	`- Dispose (p) releases memory space of an identified variable although a reference to the variable still exists.`
	
	`- When Dispose (p) is called, the value of p is undefined.`
	
	`- Prior to the Dispose (p) call, p↑ was created by New (q, c1, ..., cn) or New (q, c1, ..., cn, e) or New (q, e).`
	
	`- Prior to the Dispose (p, k1, ..., km) call, the identified variable p↑ was created by means of New (q, c1, ..., cn), where m is not equal to n.`
	
	`- With Dispose (p, c1, ..., cn) or Dispose (p, c1, ..., cn, e), the identified variable p↑ has active variants other than those specified by the CASE constants c1 to cn.`
	
	`- Prior to the Dispose (p, e) call, p↑ was created by means of New (q, a), where a is not equal to e. The same applies by analogy to Dispose (p, c1, ..., e) and New (q, k1, ..., kn, a).`
	
	`- With Dispose (p, e) or Dispose (p, c1, ..., cn, e), the value of e does not lie in the value range of the index type of the corresponding ARRAY type, or it is less than 1 or greater than the maximum length of the corresponding variable string type.`

Dispose with reduced memory areas

Dispose (p, e)

The effect of this statement is analogous to that of Dispose (p).

The domain type of p must be an ARRAY type, a variable string type, or a RECORD type with an ARRAY type or variable string type for the final field in its field list.

With an ARRAY type, the expression e must be assignment-compatible with the index type of the ARRAY type.
With a variable string type, the expression e must have an Integer type.
The value of e must not be less than 1 or greater than the maximum length of the variable string type.

The Pointer expression p must point to an identified variable p↑ which was created by means of New (q, e). The same value e must be specified in all cases.

Example

```
TYPE
   t : ARRAY [1..100] OF Real;
VAR
   p : ↑t;
BEGIN
   New (p, 45);
   ... { processing }
   Dispose (p, 45);
END
```

Dispose (p, c1, ..., cn)

The effect of this statement is analogous to that of Dispose (p).

The domain type of p must be a RECORD type. It has nested variants to which the CASE constants c1, ..., cn belong. These CASE constants must be enumerated in the sequence defined by the nesting of the variant parts. Variants which are not listed must be located in a nesting level deeper than cn.

The Pointer expression p must point to an identified variable p↑ which was created by means of New (q, c1, ..., cn). The same CASE constants must be specified in all cases.

Example

Given the following data structure t, consisting of nested variants which are referred to in the examples below.

```
TYPE
  char_range = ’0’..’1’;
  t = RECORD
         t1: Integer;
         CASE t2: Boolean OF
            False:
               (t21: Real;
                CASE t22: char_range OF
                   ’0’: (t221: ARRAY [0..10] OF Integer);
                   ’1’: (t222: Short_Integer));
            True: (t23: Char);
      END;
VAR
   p, q : ↑t;
BEGIN
   New (p, False);
   New (q, False, ’0’);
   ...  {processing}
   Dispose (q, False, ’0’);
   Dispose (p, False);
END;
```

Dispose (p, c1, ..., cn, e)

The effect of this statement is analogous to that of Dispose (q).

The domain type of the FILE type f must be a RECORD type with variants to which the CASE constants c1, ..., cn belong. These CASE constants must be enumerated in the sequence defined by the nesting of the variant parts.

The field list of the variant belonging to the final CASE constant cn must not have any further nested variants. The final field in the field list of the innermost variant must have an ARRAY type or a variable string type.

With an ARRAY type, the expression e must be assignment-compatible with the index type of the ARRAY type. With a variable string type, the expression e must have an Integer type. The value of e must not be less than 1 or greater than the maximum length of the variable string type.

The Pointer expression p must point to an identified variable p↑ which was created by means of New (q, c1, ..., cn, e).

Example

This example refers to definitions given in the preceding example.

```
New (p, False, ’0’, 2);
...
Dispose (p, False, ’0’, 2);
```

Mark (p)

Mark is used to store the current heap level in the variable p of the generic pointer type Pointer. The heap is now marked by the pointer p at its current position. Additional objects created with New will be initialized beyond this heap mark. Release (p) can be used to reset the heap level to this value. This releases the memory space of all identified variables which were created with New following the associated Mark (p) call.

Example

```
VAR
   start_level : Pointer;
BEGIN
   Mark (start_level);
   WHILE condition DO BEGIN
      ...
      New (...);
      ...
      END;
   Release (start_level);
END;
```

New (p)

The parameter p must be a variable access of a Pointer type other than the generic (and therefore private) pointer type.

New (p) requests space on the heap for a new identified variable of the domain type of p. Following New (p), p points to this identified variable p↑, whose value is still undefined.

Although the parameter p must be a variable access, it is not an actual variable parameter. In particular, p may also be a component of of packed RECORD or ARRAY.

The New procedure also offers the possibility of minimizing memory requirements. For RECORD types with variants, the requisite combination of variants can be specified so that space is made available for these variants only. For ARRAY types or variable string types, or for RECORD types containing an ARRAY type or variable string type as their final component, it is possible to specify the maximum index value for which memory space is to be made available.

Possible runtime errors:

The table below lists all possible errors which may arise in conjunction with New (p) or New with reduced memory request.

`Memory_Error`	`- The memory space required for the identified variable to be created is not available.`
`unpredictable effects`	`- An identified String variable which was created with New (p, e) or which is the final component of an identified String variable created with New (p, c1, ..., cn, e) is assigned a character string longer than e.`
	`- In an identified variable created with New (p, c1, ..., cn), a variant is activated other than the one specified by the CASE constants c1 to cn.`
	`- An identified variable created by means of New (p, c1, ..., cn), New (p, e) or New (p, c1, ..., cn, e) appears as a whole in an expression or as the left-hand side of a value assignment, or is passed as a parameter.`
	`- With New (p, e) or New (p, c1, ..., cn, e), the value of e does not lie in the value range of the index type of the corresponding ARRAY type, or it is less than 1 or greater than the maximum length of the corresponding variable string type.`

New (p, e)

The effect of this statement is analogous to that of New (p). The memory space may be requested in abbreviated form.

The domain type of p must be an ARRAY type, a variable string type, or a RECORD type with an ARRAY type or variable string type for the final field in its field list.

With an ARRAY type, the expression e must be assignment-compatible with the index type of the ARRAY type.
With a variable string type, the expression e must have an Integer type and its value must not be less than 1 or greater than the maximum length of the variable string type.

The amount of memory space allocated is the same as that allocated for the components m to n in the case of an ARRAY [m..n], or for the components 1 to e in the case of a character string.

New (p, c1, ..., cn)

The effect of this statement is analogous to that of New (p). The memory space may be requested in abbreviated form.

The domain type of p must be a RECORD type. It must contain nested variants to which the CASE constants c1, ..., cn belong. These CASE constants must be enumerated in the sequence defined by the nesting of the variant parts. Variants which are not listed must be located in a nesting level deeper than cn.

For the combination of variants thus defined, an identified variable is created with exactly this memory requirement.

The value of p↑ is undefined; in particular, no corresponding tag fields have yet been initialized. Thus, no variant has yet been activated.

The combination of variants for a variable created in this manner must not be changed by assigning conflicting values to the selector fields.

Example

```
TYPE
   r = RECORD
          ri : Integer;
          CASE rb : Boolean OF
             True  : (rta : ARRAY[1..10] OF Char);
             False : (rfr : Real;
                     CASE rfc : Char OF
                        'A': (rfai : Integer);
                        'B': (rbfr : Real);
                        ELSE: ();
                     );
       END;

VAR
   p : ↑r;

BEGIN
   New (p, True);
   { creates an object with the fields ri, rb, rta }
   New (p, False, 'A');
   { creates an object with the fields ri, rb, rfr, rfc, rfai }
END;
```

New (p, c1, ..., cn, e)

The effect of this statement is analogous to that of New (p). The memory area may be requested in abbreviated form.

The domain type of p must be a RECORD type with variants to which the CASE constants c1, ..., cn belong. These CASE constants must be enumerated in the sequence defined by the nesting of the variant parts.

The field list of the variant belong to the final CASE constant cn must not contain any further nested variants.

The final field of the field list of the innermost variant must have an ARRAY or variable string type.

With an ARRAY type, the expression e must be assignment-compatible with the index type of the ARRAY type.
With a variable string type, the expression e must have an Integer type and its value must not be less than 1 or greater than the maximum length of the variable string type.

Example

```
TYPE
   r = RECORD
          ri : Integer;
          CASE rb : Boolean OF
             TRUE  : (rta : ARRAY[9..17,1..10] OF Char);
             False : (rfr : Real;
                     CASE rc : Char of
                        'A': (rfa : Integer);
                        'B': (rfb : String [26]);
                        ELSE: ();
                     );
       END;

VAR
   p : ↑r;

BEGIN

  New (p, True, 11);
              { generates an object with the fields ri, rb,
                rta [9..11, 1..10] }

  New (p, False, 'B', 20);
              { generates an object with the fields ri, rb,
                rfr, rc, rfb [1..20]  }
END.
```

Note for New with abbreviated memory request

Identified variables created by New (p, e), New (p, c1, ..., cn) or New (p,c1, ..., cn, e) must not occur as a whole in expressions or as the left-hand side of a value assignment, and they must not be passed as parameters. This restriction must be made as, in the above-named cases, the variable is described or read as a whole. When variables are set up in abbreviated form, this causes the contiguous memory space to be overwritten, or non-allocated memory space to be accessed.

Release (p)

The parameter p must be an expression of the generic pointer type Pointer.

The value of p must have arisen by calling Mark.

The heap is reset to the level specified in p. The memory space is released for all identified variables created with New since the associated Mark (p) call. Similarly, all identifying values created with Mark (q) since the Mark (p) call are destroyed, and cannot be used in further Release (q) calls.

Unlike Dispose, Release can be used to release more than one object on the heap.

Possible runtime errors:

Pointer_Error	- When Release (p) was called, the identifying value of p was not created by a Mark call.
unpredictable effects	- Release (p) releases the memory space of an identified variable although there still exists a reference to this variable. - The identifying value passed with the Release (p) call was destroyed by another Release (q) call.

Example

The two statement sequences

```
Mark (p);
New  (p1);
New  (p2);
New  (p3);
process (p1, p2, p3);
Release (p);
```

and

```
New  (p1);
New  (p2);
New  (p3);
process (p1, p2, p3);
Dispose (p1);
Dispose (p2);
Dispose (p3);
```

are equivalent.

String Processing Subprograms

Concat (s1, s2, ..., sn)

The Concat function is used to combine any number of character strings into a single character string. The character string si is stored consecutively from left to right in the sequence specified.

s1, ..., sn must be character string expressions.

```
   Concat (s1, s2, ..., sn)
```

`with n >= 2 is equivalent to`

```
   Concat (s1, Concat (s2, ..., sn)).
```

Concat (s1) returns the value s1 itself, and has a variable string type whose maximum length is the length of the character string s1, regardless of the type of s1.

Concat (s1, s2) returns a character string whose length is Length (s1) + Length (s2).

If s1, ..., sn are static expressions, then a Concat (s1, ..., sn) call is likewise a static expression.

Note

The sum of Maxlength (s1) +...+ Maxlength (sn) must be less than or equal to the maximum permissible string length (2^{15}-1).

Example

```
CONST
   s1 = ’This is’;
VAR
   s2 : String;
BEGIN
   s2 := ’sample string.’;
   s2 := Concat (s1, ’ a ’, s2);
   { s2 = ’This is a sample string.’ }
END.
```

Delete (s, i, n)

The Delete procedure is used to delete n characters from position i in the generalized String variable s and to shift the remaining characters to the left. s must be of a variable string type. i and m must be Integer-type expressions for which $i \geq 1$, $n \geq 0$, $i+n-1 \leq$ length (s).

Possible runtime errors:

`String_Error`	`- With Delete(s,i,n), i<1 or n<0 or (i+n-1) > Length(s).`

Example

```
VAR
   a : String;
BEGIN
   a := 'This is a string deletion.';
   Delete(a,11,7);
   { a now has the value 'This is a deletion.' }
END.
```

Insert (s1, s2, i)

The Insert procedure inserts the character string s1 in the generalized String variable s2 from position i.

s1 must be a character string expression. s2 must be a String-type variable. i must be an expression whose value is assignment-compatible with Short_Integer.

If i is greater than the actual length of s2, then s2 and s1 will be concatenated.

Possible runtime errors:

String_Error	- With Insert (s1, s2, i), i<1 or Length (s2) + Length (s1) > Maxlength (s2).

```
Insert (s1, s2, i) with i ≤ length(s2) is equivalent to

   s2 := Concat (Substring (s2, 1, i-1),
                 s1,
                 Substring (s2, i, Length (s2)-i+1) )

Insert (s1, s2, i) with i > length (s2) is equivalent to

   s2 := Concat (s2, s1)
```

Example

```
   PROGRAM example;
   VAR
      b : String [10];
      c : String [20];

   BEGIN
      b := ' is ok';
      Insert ('The string', b, 1);  { causes String_Error }

      c := b;
      Insert ('The string', c, 1);  { correct call }
      { c now has 'The string is ok' }

      Insert ('ay', c, 100);        { correct call }
      { c now has 'The string is okay' }
    END.
```

Length (s)

The function call Length (s) returns the length of the character string expression s as a value of the type Short_Integer.

If s is of a variable string type, Length (s) returns the actual length of the value of s.

If s is of a fixed string type, Length (s) returns the number of the values in the index type of s.

If s is of type Char, Length (s) returns the value 1.

If s is a static expression or the type of s is an ARRAY type (fixed string type) or the type Char, the Length (s) call is likewise a static expression.

Example

```
PROGRAM example;
CONST
   letters =’abc’;

VAR
   ispacked : PACKED ARRAY [1..5] OF Char;
   string10 : String [10];
   len      : Integer;
BEGIN
   string10 := ’;’;

   len    := length (string10);    { len = 1 }

   len    := length (letters);     { len = 3 }

   len    := length (ispacked);    { len = 5 }

   len    := length (’’);          { len = 0 }
END.
```

Position (s1, s2)

The Position function can be used to determine whether, and if so where, the character string s1 is contained in the character string s2. s1 and s2 must be character string expressions.

The Position (s1, s2) call returns the value 0 if character string s1 does not occur as a substring in character string s2, or if s1 has the type Char and the character s1 does not occur in character string s2. Otherwise, Position (s1, s2) returns the least positive Integer value i such that

s1 = Substring (s2, i, Length (s1)).

Position returns the value 1 when s1 is the empty string.

s1 and s2 must be character string expressions. The result type of Position is Short_Integer.

Example

```
PROGRAM example;
VAR
   a   : String;

   pos : Integer;

BEGIN
   a   := ’ship’;

   pos := position (’ship’, a);         { pos = 1 }

   pos := position (’h’, a);            { pos = 2 }

   pos := position (’’, a);             { pos = 1 }

   pos := position (a, ’relationship’); { pos = 9 }

   pos := position (a, ’x’);            { pos = 0 }
END.
```

Readstring (s, v1, ..., vn)

The Readstring procedure reads values from the character string expression s to the variable accesses v1, ..., vn. The procedure works in the same way as the Read procedure for textfiles, except that instead of the FILE variable there has to be a character string expression.

s must be a character string expression, and the variable accesses v1, ..., vn must have one of the following types:

- the type Char or a subrange thereof,
- an Integer type or a subrange thereof,
- a Real type,
- a variable string type,
- a fixed string type.

The call

Readstring (s, v1, ..., vn)

is then equivalent to the compound statement

```
BEGIN
   Rewrite (f); Writeln (f, s);
   Reset (f);   Read (f, v1, ..., vn)
EXCEPTION
   IF Error_Number = Eof_Error THEN
      Raise (String_Error)
   ELSE
      Raise (0);
END
```

where f is a FILE variable of type Text which is used nowhere else in the program. v1, ..., vn are subject to the same conventions as the parameters of the Read procedure (see section 15.1). The equivalent compound statement is not the final result, but is used for purposes of explanation.

Note

If vi is a String-type variable, then all the remaining characters in character string expression s are consumed by vi (provided the maximum length of s is sufficient; otherwise a String_Error will occur).
If vi is followed by further parameters, these parameters cannot be assigned further values, and a String_Error is reported.

Possible runtime errors:

`Numeric_Error`	- The value of the input Real number can be represented internally, but it lies outside the value range of the read parameter v.
`Read_Error`	- While an Integer number or Real number is being read from a character string expression, the following happens: - the number is syntactically incorrect or - the value of the input number is too large, thus it cannot be represented internally. With Integer numbers the value lies outside the range Long_Minint .. Long_Maxint; with Real numbers the value lies outside the range -Long_Maxreal .. Long_Maxreal.
`Range_Error`	- The value of the input Integer number can be represented internally, but it lies outside the range of the read parameter v.
`String_Error`	- With Readstring (s, v), the maximum length of the generalized String variable v is less than the length of the input character string. - With Readstring (s, v1, ..., vn), the character string expression s does not contain as many characters as is required by the read parameters v1, ..., vn.

Cross-references

Read: 15.1

Substring (s, i, n)

The function call Substring (s, i, n) returns a character string of length n, whose k-th element is the (i+k-1)-th string element of s, i.e. the substring of s of length n beginning at i.
s must be a character string expression. i and n must be Integer-type expressions for which $i \geq 1$, $n \geq 0$ and $i+n-1 \geq$ Length (s). The result type of Substring is a variable-length string type.

If s, i, n are static expressions, the Substring (s, i, n) call is likewise a static expression.

Possible runtime errors:

`String_Error`	`- With Substring(s,i,n), i<1 or n<0 or (i+n-1) > Length(s).`

Example

```
VAR
   a : String;
BEGIN
   a := 'processing package';
   a := Substring (a, 7, 4);
   { a now has the value 'sing' }
END.
```

Writestring (s, a1, ..., an)

The Writestring procedure writes the values of the expressions a1, ..., an to the String variable s. Thus, Writestring works in the same way as the Write procedure for textfiles, except that a String variable s is specified instead of the FILE variable. s must be a String-type variable, and each expression ai (i=1, ..., n) must have one of the following types:

- the type Char or a subrange thereof,
- an Integer type or a subrange thereof,
- a Real type,
- a generalized string type,
- the Boolean type.

For each expression ai, a format denoter can be specified, just as with the write parameters for Write (see section 15.1).

The call
  Writestring (s, a1, ..., an)

is then equivalent to the compound statement

```
BEGIN
   Rewrite (f); Writeln (f, a1, ..., an);
   Reset (f);   Read (f, s);
END
```

where f is a FILE variable of type Text which is used nowhere else in the program. a1, ..., an are subject to the same conventions as the parameters of the Write procedure (see section 15.1). The equivalent compound statement is not the final result, but is used for purposes of explanation.

Possible runtime errors:

String_Error	- With Writestring (s, a1, ..., an), the maximum length of the String variable s is smaller than the character string formed from the write parameters a1, ..., an.
Range_Error	With Writestring (s, a:n), the total output length n < 1 or n < 0, provided a has a variable string type. - With Writestring (s, a:n:m), the total output length n < 1 or the number of digits following the decimal point m < 1.
unpredictable effects	- With Writestring (s, p1,..., pn), one of the write parameters p1,...,pn contains a reference to the String variable s.

Cross-references

Write:	15.1

Arithmetic Functions

The arithmetic functions are used for numeric calculations. The type of the parameter may be a Real type or an Integer type. The result type is derived from the type of the parameter.

The following applies to the functions Abs and Sqr:

Type of parameter	Type of result
Integer	Integer
Real	Real
Short_Integer	Integer
Long_Integer	Long_Integer
Short_Real	Real
Long_Real	Long_Real

Table 15-1 Result types for Abs and Sqr

For the remaining arithmetic functions, the following applies:

Type of parameter	Type of result
Integer	Real
Real	Real
Short_Real	Short_Real
Long_Real	Long_Real
universal Real type	universal Real type
Integer type	universal Real type

Table 15-2 Result types of arithmetic functions

The universal Real type is adapted to its context; see section 9.3.1.

When the Abs and Sqr functions are called with a static expression as actual parameter, the calls are themselves static expressions. Calls of the other standard arithmetic functions are not static expressions.

Arithmetic functions are calculated with at least the level of precision of the result type; however, they may also be calculated with a higher level of precision.
When the result type is the universal Real type, calculation takes place with at least the precision of Long_Real.

Abs(x)

calculates the absolute value of x.

Possible runtime errors:

`Numeric_Error`	- With `Abs(x)`, the function result does not lie in the value range of the result type (`Integer` or `Long_Integer`, `Short_Real`, `Long_Real`).

Arctan(x)

calculates the main value of the arc tangent of x in radians.

Cos(x)

calculates the cosine of x. Here x must be specified in radians.

Exp(x)

calculates `e**x`, where e is the base of the natural logarithm.

Possible runtime errors:

`Numeric_Error`	- The result of `Exp(x)` does not lie in the value range of the result type.

Ln(x)

calculates the natural logarithm of x.

Possible runtime errors:

`Numeric_Error`	`- With Ln(x), x <= 0.`

Sin(x)

calculates the sine of x. Here x must be specified in radians.

Sqr(x)

calculates the square of x (i.e. x*x).

Possible runtime errors:

`Numeric_Error`	`- With Sqr(x), the function result does not lie in the value range of the result type (Integer or Long_Integer, Short_Real, Long_Real).`

Sqrt(x)

calculates the square root of x.

Possible runtime errors:

`Numeric_Error`	`- With Sqrt(x), x < 0.`

Transfer Functions

When a transfer function is called with a static expression as its actual parameter, the call is itself a static expression.

Long (x)

transfers the value of expression x of type Short_Integer to the same value of type Long_Integer.

Round(x), Short_Round(x), Long_Round(x)

For the expression s, which must be of type Real, these functions return a result of the corresponding Integer type:

– The result type for Round is Integer,
– the result type for Short_Round is Short_Integer and
– the result type for Long_Round is Long_Integer.

The value of Round(x) is defined as follows:

– `Round(x) = Trunc(x + 0.5) for x ≥ 0,`

– `Round(x) = Trunc(x - 0.5) for x < 0.`

Possible runtime errors:

Numeric_Error	- The result of Round(x) or Short_Round(x) or Long_Round(x) does not lie in the value range of the result type (Integer or Short_Integer or Long_Integer).

Example

```
Round(3.5)   yields  4
Round(-3.5)  yields -4
```

Trunc(x), Short_Trunc(x), Long_Trunc(x)

calculates the integral part of x.

For the expression x, which must be of a Real type, these functions return a result of the corresponding Integer type:

– The result for Trunc is Integer,
– the result for Short_Trunc is Short_Integer, and
– the result for Long_Trunc is Long_Integer.

The value of Trunc (x) is defined by the following conditions:

– Trunc(x) is integral,

– $0 \le$ x-Trunc(x) < 1 for $x \ge 0$,

– $-1 <$ x-Trunc(x) ≤ 0 for $x \le 0$.

Possible runtime errors:

`Numeric_Error`	`- The result of Trunc(x) or Short_Trunc(x) or Long_Trunc(x) does not lie in the value range of the result type (Integer or Short_Integer or Long_Integer).`

Example

```
Trunc(3.5)   yields  3
Trunc(-3.5)  yields -3
```

Ordinal Functions

When an ordinal function is called with a static expression as its actual parameter, the ordinal function is itself a static expression.

Card (s)

returns the number of members contained in the set determined by the value of the expression s, where s must have the type SET. The result is of type Integer.

Chr (x)

For the Integer-type expression s, this function returns that Char-type value which is encoded by the value of x.
For each Char-type value c the following applies: Chr (Ord (c)) = c

`Range_Error`	`- The character value Chr(x) does not lie in the value range of the type Char.`

Note

The character set of a Pascal-XT processor is implementationdefined (see also section 6.2.3). If a program is written for different Pascal-XT implementations, no further assumptions may be made regarding the assignment of Integer values to character values and vice versa.

Cross-references

Char: 6.2.3

Ord (x)

For the expression x, which must have an ordinal type, this function returns that Integer-type number which is defined as the ordinal number of the value of expression x.

Cross-references

Char:	6.2.3
Enumerated type:	6.2.5

Pred(x)

For the expression x, which must have an ordinal type, this function returns the value of the same type whose ordinal number is 1 less than that of expression x (predecessor).

Possible runtime errors:

`Range_Error`	`- The result of Pred (x) does not lie in the value range of the type of x.`

Setmax (s)

Setmax (s) returns the value of the greatest member contained in the set which is determined by the value of expression s, which must be of a SET type. The result type is the base type of the SET type of s. The empty set (i.e. "[]") must not be specified as a parameter.

Possible runtime errors:

`Set_Error`	`- With Setmax (s), the value of expression s is equal to the empty set (see section 9.4).`

Setmin (s)

Setmin (s) returns the value of the least member contained in the set which is determined by the value of expression s, which must be of a SET type. The type of the result is the base type of the SET type of s. The empty set (i.e. "[]") must not be specified as a parameter.

Possible runtime errors:

`Set_Error`	`- With Setmin (s), the value of expression s is equal to the empty set (see section 9.4).`

Cross-references

Sets:	6.3.4
Set constructors:	9.4

Succ (x)

For the expression x, which must have an ordinal type, this function returns the value of the same type whose ordinal number is 1 more than that of expression x (successor). The result type is the base type of the SET type of s. The empty set (i.e. "[]") must not be specified as a parameter.

Possible runtime errors:

`Range_Error`	`- The result of Succ(x) does not lie in the value range of the type of x.`

Boolean Functions

The Boolean functions Eof and Eoln were already discussed in section 15.1 (file proces-sing).

Odd (x)

This function returns the Boolean value True for an Integer-type expression x if x is odd, and the Boolean value False if x is even.
When the Boolean function Odd is called with a static expression as its actual parameter, the function is itself a static expression.

Transfer Procedures

The transfer procedures Pack and Unpack make it possible to transfer components between packed and unpacked ARRAYs which have the same component type.
In Pascal-XT, components may also be transferred between variable strings and unpacked ARRAYs with the component type Char.
In particular, it is possible to transfer only a subrange.

Both procedures have the parameters z, a and i, for which the following rules apply:

z	is of type PACKED ARRAY [s2] OF t or of type String[n]
a	is of type ARRAY [s1] OF t
i	is an expression whose value is assignment-compatible with s1.

If z is of type PACKED ARRAY, then z and a must have the same component type. t must not be a FILE type or contain a component of a FILE type.
If z has the type String[n], the component type t of a must be the type Char.

Pack (a, i, z)

The Pack procedure transfers components of an unpacked ARRAY-type expression to a variable of a packed ARRAY or String type.
In Standard Pascal, a must be a variable access.

Packing an ARRAY

From an unpacked ARRAY-type expression a, starting at index i, Pack transfers as many components to variable z as will fit into that variable. a must contain at least as many components from index i as are to be transferred, i.e. the following rule must hold:

```
Last(s1) - i + 1 >= Last(s2) - First(s2) + 1
```

Packing a string

From an unpacked ARRAY expression a, Pack transfers all characters from a[i] to a[Last(s1)] to the String variable z. The number of characters transferred must be less than or equal to the maximum length of z, i.e. the following rule must hold:

Last(s1) - i + 1 <= Maxlength(z)

Possible runtime errors:

`Index_Error`	- With Pack (a,i,z), the value of expression i does not lie in the value range of the index type of the unpacked ARRAY parameter a. - With Pack (a,i,z), the index range of the unpacked array a is exceeded when the components of a are transferred to the packed array z starting at index i.
`String_Error`	- With Pack (a,i,z), the maximum length of the String variable z is too small to include all characters from the unpacked array a starting at index i.
unpredictable effects	- With Pack (a,i,z), a component accessed in the unpacked array a is undefined.

Examples

In the example below, the components a[11] to a[20] are transferred to the variable p.

```
TYPE
   t  = ARRAY [-40..40] OF Boolean;
   tp = PACKED ARRAY [1..10] OF Boolean;
VAR
   a : t;
   p : tp;
BEGIN
   Pack (a, 11, p);
END
```

The following program fragment transfers the components a[17] to a[35] to the String variable s.

```
TYPE
   t = ARRAY [-10..35] OF Boolean;
VAR
   a : t;
   s : String;
BEGIN
   Pack (a, 17, s);
END
```

Cross-references

PACKED:	6.3
ARRAY types:	6.3.1
Variable string types:	6.3.2.2
First, Last:	15.9
Maxlength:	15.9

Unpack (z, a, i)

The Unpack procedure transfers data from a packed ARRAY or string to a specifiable range of an unpacked ARRAY.
In Standard Pascal, z must be a variable access.

Unpacking a packed ARRAY

Unpack transfers all components from the package ARRAY expression z to the pakked ARRAY variable, starting at a[i]. The number of components in ARRAY a from i to Last (s1) must be greater than or equal to the number of components in the pakked ARRAY z, i.e. the following rule must hold:

Last(s1) - i + 1 >= Last(s2) - First(s2) + 1

Unpacking a string

Unpack transfers all characters in the variable string expression z to the unpacked ARRAY variable a, starting at a[i]. The number of characters transferred is equal to the actual length (Length (z)) of expression z, and the following rule must hold:

Last(s1) - i + 1 >= Length(z)

Possible runtime errors:

`Index_Error`	- With Unpack (z,a,i), the value of expression i does not lie in the value range of the index type of the unpacked array parameter a. - With Unpack (z,a,i), the range in the unpacked array, starting at index i, is too small to include all the components of packed array z. - With Unpack (z,a,i), the character string expression z contains more characters than can be transferred to the unpacked array a starting at index i.
`unpredictable effects`	- With Unpack (z,a,i), some component of the packed array z is undefined.

Examples

The components a[4] to a[11] are replaced by the components z[3] to z[10].

```
TYPE
   tz = PACKED ARRAY [3 .. 10] OF Integer;
   ta = ARRAY [0 .. 39] OF Integer;
VAR
   z : tz;
   a : ta;
BEGIN
   z := tz (1, 2, 3, 4, 5, 6, 7, 8);
   unpack (z, a, 4);
END
```

The components c[13] to c[17] are replaced by the five characters
('H','e','l','l','o') of the String variable s.

```
TYPE
   tc = ARRAY [10..60] OF Char;
VAR
   c : tc;
   s : String;
BEGIN
   s := 'Hello';
   Unpack (s, c, 13);
END
```

Cross-references

PACKED:	6.3
ARRAY types:	6.3.1
Variable string types:	6.3.2.2
Length:	15.3
First, Last:	15.9

Attribute Functions

Attribute functions primarily return information on types. Thus, the argument of an attribute function is generally a type identifier. However, a variable is also permitted as an argument. In this case, the attribute functions return the values of the attributes of the type of the variable.

The information refers to

– alignment (Alignof),
– number of bits for representing ordinal-type values (Bitsizeof),
– the least value of an ordinal type (First),
– the greatest value of an ordinal type (Last),
– the maximum length of a variable string type (Maxlength),
– the offset of fields in a record (Offsetof) and
– the memory requirement (Sizeof).

Attribute functions are always static expressions, except for Sizeof (t, e) and Sizeof (t, c1, ..., cn, e) if the expression e is not static.

In the case of the functions First and Last, the result type is the same type as that of the actual parameter. With the other functions, it is Short_Integer if the value of the result lies within the value range of Short_Integer. Otherwise, the result type is Long_Integer.

With the non-static calls Sizeof (t, e) or Sizeof (t, c1, ..., cn, e), the result type is same as that of a Sizeof call without the final parameter e.

Implementation-defined characteristic

The size of a storage unit is implementation-defined. It may be 1 byte or a multiple thereof (generally a power of 2).

Note

If a program has interfaces to program parts written in other languages, the attribute functions may be used to obtain information on how the data to be transferred are represented in memory.

Cross-references

Memory representation:	6.3.3.2
Attributes:	6.7

Alignof (t)

Alignof returns the memory alignment required for variables of type t. The alignment is a positive integer, normally a power of 2, and derives from the manner in which the processor is allowed to access memory space. The The address of each variable of type t is an integral multiple of Alignof (t).

Bitsizeof (t)

The function Bitsizeof (f) returns the minimum number of bits which are normally required to represent the values of t. The type t must be an ordinal type.

Example

```
TYPE
   t1 = (c1, c2, c3);
   t2 = (k1, k2, k3, k4);
   r = PACKED RECORD
       a (0 : 0..bitsizeof (t1)) : t1;
       b (0 : bitsizeof (t1)..7) : t2;
       END;
```

First (t)

returns the least value of the ordinal type t.

Last (t)

returns the greatest value of the ordinal type t.

Example

```
VAR
   i : ordinal_type;
BEGIN
   ...
   FOR i := First (ordinal_type) TO Last (ordinal_type) DO ... ;
   ...
END
```

The statement given below would have the same effect. In this case, the values of First and Last are derived from the type of i.

```
BEGIN
   ...
   FOR i := First (i) TO Last (i) DO ... ;
   ...
END
```

Maxlength (t)

returns the maximum length of the variable string type t.

Example

```
BEGIN
   ...
   IF Length (s1) + Length (s2) > Maxlength (s) THEN
      error_handling
   ELSE
      s := Concat (s1, s2);
   ...
END
```

Offsetof (t, f)

This function returns the offset of a RECORD field relative to the start of the RECORD, in number of memory units. t indicates any RECORD type of a variable of any RECORD type. f is a field identifier which is closest-contained in t, i.e. f must not be contained in a nested RECORD.
The size of a memory unit is implementation-defined (see Appendix A.6).

Sizeof (t)

Sizeof (t) returns the number of memory units required to represent values of type t. The size of a memory unit is implementation-defined (see Appendix A.6).

Sizeof (t, e)

Sizeof (t, e) returns the number of memory units occupied for an identified variable p↑ of type t in conjunction with a New (p, e) call.

Type t must be an ARRAY type, a variable string type, or a RECORD type with an ARRAY type or variable string type for the final field in its field list.

With an ARRAY type, the expression e must be assignment-compatible with the index type of the ARRAY type.
With a variable string type, the expression e must have an Integer type, and its value must not be less than 1 or greater than the maximum length of the variable string type.

Sizeof (t, c1, ..., cn)

Sizeof (t, c1, ..., cn) returns the number of memory units which are occupied for an identified variable p↑ of type t in conjunction with a New (p, c1, ..., cn) call.

Type t must be a RECORD type. It must contain nested variants to which the CASE constants c1, ..., cn belong. These CASE constants must be enumerated in the sequence defined by the nesting of the variant parts.
Variants which are not listed must be located in a nesting level deeper than cn.

Sizeof (t, c1, ..., cn, e)

Sizeof (t, c1, ..., cn, e) returns the number of memory units which are occupied for an identified variable p↑ of type t in conjunction with a New (p, c1, ..., cn, e) call.

Type t must be a RECORD type with variants to which the CASE constants c1, ..., cn belong. These CASE constants must be enumerated in the sequence defined by the nesting of the variant parts.

The field list of the variable belonging to the final CASE constant cn must not contain any further nested variants. The final field in the field list of the innermost variant must have an ARRAY type or a variable string type.

With an ARRAY type, the expression e must be assignment-compatible with the index type of the ARRAY type.
With a variable string type, the expression e must have an Integer type, and its value must not be less than 1 or greater than the maximum length of the variable string type.

Example of using attribute functions

```
...
TYPE
   color = (red,yellow,green,blue,brown,black);
   data  = PACKED RECORD
           tint (0: 0..Bitsizeof(color)-1): color;
           ...
           END;
   bytes = PACKED ARRAY [1..Sizeof(data)] OF Char;
...

VAR
   f : color;
   d : data;
   b : bytes;
...

BEGIN
   ...

   FOR f := First (color) TO Last (color) DO ...

   ...
END
```

Unchecked Type Conversion

Pascal-XT has a function for unchecked type conversion between values with the same memory representation.

Convert (x, t)

For the expression s and the type identifier t, this function returns that value of type t which has the same representation in memory as the value x. Type t must not be a FILE type, and it must not contain any FILE-type components. If both type t and the type of x are ordinal types, the following rule must hold:

Ord (First (t)) <= Ord (x) <= Ord (Last (t)),

For all other type combinations, the following must hold:

Sizeof (x) = Sizeof (t).

If the expression x is a static expression of a scalar or Pointer type, and if type t is likewise a scalar type or a Pointer type, the Convert function call is also a static expression.

Possible runtime errors:

`unpredictable effect`	`With Convert (x, t), the representation in memory of x is not a permissible value of t.`

Note

It is easy to see that this function is a back door for abandoning the Pascal programming style. However, it does have a certain justification when solving problems in a machine-bound environment (e.g. for directly programming the buffering of data in mass storage units). Be forewarned about using the type concept when solving problems at the user level!

Exception Handling Subprograms

Error_Number

This parameterless function returns the error number (see section 14.2) of the most recent exception (error) situation. Its value is 0 if no exception situation has occurred.

Raise (n)

Raise can be used to create a programmed exception (error) situation with error number n or to propagate that exception (error) situation within an exception handling part. n must be an Integer-type expression with the following meaning:

n < 0:	The negative error numbers are reserved for the Pascal-XT system. For the values -1 to -16 there exist required identifiers (see sections 14.1, 5.2).
n = 0:	Any exception which occurred previously will be propagated (forwarded) without changing the error number and without losing any information regarding the original location of the error (unlike Raise (Error_Number)). A System Error occurs if there was no exception situation prior to calling Raise (0). Thus, n = 0 does not represent an error number.
n > 0:	The positive error numbers may be used as you wish.

If n <> 0, the effect is analogous to the situation when a "genuine" error occurs, i.e. in place of the Raise call the exception with number n is created. If an error cannot be handled in an exception handling part it can be propagated (forwarded) with Raise in either of two ways:

1) Raise (Error_Number)
The most recent error is triggered again. The line with the Raise call represents the new error location and the information on the original error location is thus lost.

2) Raise (0)
The most recent error is propagated without losing the information on the original error location.

To propagate errors Raise (0) should always be used so that information on the original error location is retained for further (system-dependent) diagnostic purposes (see also examples 14-4 and 14-5 in chapter 14 and the

User’s Guide).

Possible runtime errors:

`System_Error`	`- Raise (0) was called although no error had occurred previously in this program and thus error propagation is impossible.`

Notes

- The error numbers employed in a program by a user must be unique so that in case of error the original location of the error can be unambiguously identified.
- The Pascal-XT implementations on various systems provide support when the dynamic call chain is output in case of error.
- The Pascal-XT implementations on various systems provide an error propagation procedure Reraise as part of a required package errors. To improve understanding and portability, this procedure can be used instead of Raise (0).

Cross-references

Error number, predefined exceptions:	14.1, 5.2
Exception handling:	14.3

Example of Error_Number and Raise

```
PROGRAM example (Output);
VAR
   i, j : Long_Integer;

PROCEDURE check_overflow (n,m:Long_Integer);
BEGIN
   IF (Long_Maxint-m) < n THEN
      Raise (1);
END;

BEGIN
   ....
   check_overflow (i,j):
   ....
EXCEPTION
   IF Error_Number = 1 THEN
      Writeln ('Overflow occurred')
   ELSE
      Raise (0);
END.
```

Explicit Package Initialization Procedure

Elaborate (p)

The parameter p must be a package identifier. Calling the Elaborate (p) procedure ensures that the initialization of package p will take place at the latest following the Elaborate (p) call.
If initialization has already taken place prior to the Elaborate call, the call has no effect.

Elaborate must be called only within the statement sequence of a package block. It is not allowed to occur
– within repetitive statements (WHILE, REPEAT, FOR)
– within conditional statements (IF, CASE).

Possible runtime errors:

Elab_Error	- It is not possible to continue initializing the packages of a program since otherwise loops will arise due to the use of the required procedure Elaborate.

Example

During initialization, package a uses the global variable status from package b. This variable must be initialized before being used in package b. By calling Elaborate (b) before using status in package a, you can make sure that package b is initialized.

```
PACKAGE a;
   ...

END.

WITH b;
PACKAGE BODY a;
   ...
BEGIN
   Elaborate (b);
   IF b.status THEN ...;
   ...
END.
```

```
PACKAGE b;
VAR
   status: Integer;
END.


PACKAGE BODY b;
   ...
BEGIN
   status := 0;
   ...

END.
```

Cross-references

Program execution: 13.3.3

Control Statements for the Compiler

A comment starting with a dollar sign immediately after the open comment bracket is called a pseudocomment. It contains control statements for the compiler.

```
pseudocomment      = "{$" control-statement {"," control-statement} "}".
control-statement  = option [ "=" ("On"         |
                                   "Off"        |
                                   "Restricted" |
                                   character-string) ].
option             = identifier.
```

Like comments, pseudocomments have no effect on the logical execution of a correctly written Pascal program. However, under certain conditions they may affect its physical behavior, e.g. the speed of execution, memory requirements, additional runtime error messages, the size of the compilation listing or the applicability of the debugger.

Options fall into two categories:

– Global options
  apply for the entire compilation unit. They can only be activated by control statements located before the actual beginning of the compilation unit (i.e. before the word symbols WITH, PROGRAM or PACKAGE).

– Local options
  may be activated anywhere in the program by means of control statements.

Control statements can be further subdivided according to their effect:

– Control of compilation listing (e.g. complete source listing, error list only, cross-reference listing, object code listing).

– Effect on runtime behavior (e.g. deactivation of error detection at runtime, optimization of object code).

– Generation of test tables for the symbolic debugger.

Many control statements function like "switches" for activating or deactivating options. In these cases, the value specified for the option may only be

On or Off

where On is used for activating the relevant option, and Off for deactivating it. To activate an option, it is also sufficient to specify the option alone, without "= On".

The value Restricted has only one meaning for the control statement for test table generation. Only a character string can be entered with the Title option.

Implementation-defined characteristics

The preset default values for the compiler options are implementation-defined.

Whether, and if so how, options may be activated during operation (when the compiler is called) is implementation-defined.

Examples of control statements

```
{$ List=On, Check=Off}
```

```
{$ Page}
```

Note

When developing Pascal programs you should always specify the control statements Check = On and Initialize = On. This makes it possible to detect errors in good time which might otherwise pass undetected. If neither of these two options is specified, any error which would otherwise have been detected by the options will have unpredictable effects. On the other hand, this causes additional code to be generated for the runtime checks, thereby increasing the run time of the program. For this reason you should make sure that Check = Off, Initialize = Off and Optimize = On when compiling benchmark programs for runtime measurements.

Global Options

The global options apply for the entire compilation unit, and must be specified before the keyword WITH, PROGRAM or PACKAGE.

Generate

activates or deactivates the generation of object code. Usually no code is generated when package specifications are compiled.

[= On]	Object code will be generated for this compilation unit.
= Off	No object code will be generated for this compilation unit. In this case, the local options Assembler, Check, Initialize and Optimize have no effect in this compilation unit, nor do the global options Debug and Map.

Debug

controls the generation of test tables. These test tables are needed when testing with a symbolic debugger (see section 24.1). The test tables are only generated for programs and package bodies if the Generate option is switched to On. With package specifications, the Debug option only takes effect when the package body is compiled.

If Debug = On or Debug = Restricted is entered in a package specification, all the constant, variable and type identifiers defined in it can be used in debugging statements. If the option was specified in a package body, both the declarations and line numbers of this package body as well as the declarations of the associated package specifications may be used in debugging statements.

If Debug = On or Debug = Restricted is specified, the Optimize option will have no effect.

[= On]	This compilation unit can be tested with the symbolic debugger.
= Restricted	This compilation unit can be tested to a limited extent with the symbolic debugger. Assignments are not permitted as debugging statements at test points within this compilation unit.
= Off	This compilation unit cannot be tested with the symbolic debugger.

Map

activates or deactivates the output of address tables to the compilation listing. Only if Generate = On will the Map option take effect when programs and package bodies are compiled. In the case of package bodies, the output tables also contain the objects of the associated specification.

[= On]	Address tables will be generated for this compilation unit.
= Off	No address tables will be generated.

Standard

defines whether to accept the language set of Standard Pascal or Pascal-XT.

[= On]	The language set of Standard Pascal, Level 1, will be accepted. Deviations from Standard Pascal will be reported as errors. The word symbols added in Pascal-XT may only be used as ordinary identifiers.
= L0	Same as On except that Standard Pascal, Level 0 (without conformant array schema), will be accepted. This argument can only be specified when the compiler is called.
= Off	The entire language set of Pascal-XT will be accepted.

Xref

controls output of a cross-reference listing.

[= On]	A cross-reference listing will be output containing all the identifiers used in this compilation unit. Even those identifiers will be included which are predefined (required) or which have their defining point in a foreign compilation unit.
= Off	No compilation listing will be output.

Local Options

Local options may be specified anywhere in the program. The scope is specified for each option.

Assembler

controls output of the object code listing in assembler format. When activated before or inside a statement part of a block, the option applies for the entire statement part. Deactivation of an option in a statement part only takes effect at the end of this statement part.

[= On]	Activates output of the object code listing.
= Off	Deactivates output of the object code listing.

Check

controls the generation of additional commands (Check code) for runtime error detection.

This option applies for all statements in which it is activated.

Note:

At the program development stage, you should activate the Check option for the entire compilation unit in order to detect errors in good time.

[= On]	Activates the generation of Check code.
= Off	Deactivates the generation of Check code. The error situations Numeric_Error, Range_Error, Set_Error, String_Error, Index_Error, Pointer_Error, Variant_Error and Case_Error will not necessarily be detected and may lead to unpredictable results.

Optimize

controls the optimization of object code. Optimization is performed to minimize the amount of run time. When activated before or inside the statement part of a block, this option applies for the entire statement part. Deactivation of an option in a statement part only takes effect at the end of this statement part.

[= On]	Activates optimization.
= Off	Deactivates optimization.

Initialize

controls initialization of memory areas for variables with an implementation-dependent value. Initialization of the memory area takes place

- before program execution in the case of variables in the main program block or package block,
- at the time the subprogram is called in the case of local variables in a subprogram,
- at the time New is called in the case of dynamically generated (identified) variables.

The initializing value is chosen so that, for example, an undefined identifying variable is recognized as being undefined (with Check = On). However, this value is not meant to substitute for a programmed initial assignment of a value to a variable, but is intended solely for the purpose of error location! It may happen with this option that some undefined variables are left undetected.

The Initialize option is only meaningful when the Check option is activated. At the program development stage, it is essential that both of these options be activated for the entire compilation unit.

When activated before or inside the statement part of a block, this option applies for the entire statement part. Deactivation of an option in a statement part only takes effect at the end of this statement part.

[= On]	Additional initialization code will be generated.
= Off	The memory areas will not be initialized.

List

controls output of the compilation listing.

[= On]	Activates output of the compilation listing from the beginning of the line containing this control statement.
= Off	Deactivates output of the compilation listing from the end of the line containing this statement. In this case, only errored source code lines will be listed, together with the error messages.

Page

causes a form feed to take place immediately following the line where this control statement begins.

Title = character-string

controls output of a subtitle which will be output starting from the next page of the source listing. Output of the subtitle can be terminated by entering a blank string (’’). The subtitle is entered as a character string enclosed in single quotes (apostrophes).

Example

```
{$ Title = 'Interface Description' }

{$ Title = '' }
```

The Package Concept

The package concept directly supports the principles of modularization, abstraction and information hiding. The difference between Pascal-XT and many other programming languages is that these principles are supported by packages, thereby encouraging the user of the language to apply them.

For the Pascal-XT package concept, the term "program" is taken more broadly than is intended in Standard Pascal. A program consists of a main program and any number of packages. If there are no packages, this conforms to the sole main program of Standard Pascal.

A package consists of the package specification and the package body. The specification describes **what** the package does, and the body describes **how** it does it. Specifications can be passed to a user even if he does not know the body. This is because, physically, package specification and package body are managed separately from each other.

As an example, consider the operation of an automobile. The interface is formed by the steering wheel, breaks and accelerator. These are the visible entities of the automobile. The driver does not have to know how these operating elements function. These are details which may be as complicated as you please. In a similar way, Pascal-XT offers an opportunity of hiding details in the package body.

Package specification

The specification defines the visible entities which can be addressed from other packages. These may be constants, types, variables or the headings of subprograms. Only these entities may be accessed from outside the package.

Example

```
PACKAGE complex_definitions;

TYPE complex = RECORD
                  realpart : Real;
                  imagpart : Real;
               END;

{ functions for manipulating complex numbers }
FUNCTION make_complex (r, i : Real) : Complex;
   {r becomes the real part, i the imaginary part}

FUNCTION imag_part_of (c : complex) : Real;
   {yields the imaginary part of the complex number }

FUNCTION real_part_of (c : complex) : Real;
   {yields the real part of the complex number }

{ arithmetic functions }
FUNCTION add      (left, right : complex) : complex;

FUNCTION subtract (left, right : complex) : complex;

FUNCTION multiply (left, right : complex) : complex;

FUNCTION divide   (left, right : complex) : complex;

END.
```

Package body

A package body consists of a declaration part and a statement part, each of which may be empty. If subprograms were declared in the specification, then the procedure or function identification must be specified in the package body. This does not apply to INLINE subprograms, for which the subprogram block must already be specified in the package specification, or to subprograms with a directive. In addition, constants, types, variables and subprograms may be declared which are only required within the package body and are invisible outside the package. Since specification and body form a logical unity, it follows that identifiers declared in the specification must not be declared again in the package body.

The statement part is used to initialize the package, and is processed once and only once prior to program execution (see section 13.3). The variables defined in the declaration part of a package can be stored in the statement part with initial values. By the time the first statement in the main program is executed, all the packages will already have been initialized.

A specification must be accompanied by at least one package body. Depending on boundary conditions, there may also be two or more package bodies performing the same service of the package but in different ways. For example, different versions of a package body may contain message texts in different languages. The desired package body is chosen as needed without affecting the rest of the program.

The package body for the above specification looks as follows:

```
PACKAGE BODY complex_definitions;

FUNCTION make_complex (r, i : Real) : complex;
   {r becomes the real part, i the imaginary part }
BEGIN
   make_complex := complex (r, i);
END;

FUNCTION imag_part_of (c : complex) : Real;
   {yields the imaginary part of the complex number }
BEGIN
   imag_part_of := c.imagpart;
END;

FUNCTION real_part_of (c : complex) : Real;
   {yields the real part of the complex number }
BEGIN
   real_part_of := c.realpart;
END;

{ arithmetic functions }

FUNCTION add      (left, right : complex) : complex;
BEGIN ... END;

FUNCTION subtract (left, right : complex) : complex;
BEGIN ... END;

FUNCTION multiply (left, right : complex) : complex;
BEGIN ... END;

FUNCTION divide   (left, right : complex) : complex;
BEGIN ... END;

BEGIN
END { complex }.
```

In this example no initialization is necessary. In the case of a random number generator, a start value is required. Initialization takes place as follows:

```
PACKAGE random_number_generator;

FUNCTION random : real;

END.



PACKAGE BODY random_number_generator;
VAR
   seed : integer;

FUNCTION random : Real;
BEGIN
   ...
END { random };

BEGIN { initialization }
   seed := 123456;
END.
```

The variable seed is known only within the package random_number_generator. In the statement part, the variable is assigned a start value. At the time the package_random_number generator is initialized, this assignment is processed before the program using this package is started.

Compilation units

Main programs, package specifications and package bodies are referred to as compilation units. They are managed separately and even compiled separately.

Relations between compilation units

If entities in a specification of another package are accessed in a compilation unit, the name of the package must be specified in the WITH list of the compilation unit. In this way, all entities in this package are made visible and can be used by specifying the package name and the desired identifier. The example below illustrates the use of the package complex_definitions in the program some_program.

```
WITH complex_definitions;
PROGRAM some_program
VAR
   x, y, z : complex_definitions.complex;
BEGIN
   ...
   z := complex_definitions.add (x,y);
   ...
END.
```

To avoid having to specify the package name each time, the identifiers of the packages to be imported may be specified in the USE list. Imported identifiers can be used without prefixing the package name. The package from which the identifiers are imported must be specified beforehand in a WITH list.

```
WITH complex_definitions;
FROM complex_definitions USE complex, add;
PROGRAM some_program;
VAR
   x, y, z : complex;
BEGIN
   ...
z := Add (x,y);
   ...
END.
```

The use of the USE clause is practical because it permits shorter identifiers; at times, however, it may adversely affect the clarity and readability of the program. The USE clause does not forbid prefixing the package name in order to use an identifier.

Compilations

While a compilation unit is being compiled, the compiler checks whether the entities from the packages specified in the WITH lists are used correctly, e.g. whether the parameters for subprograms are correct.

If more than one package is being compiled, a particular compilation sequence must be adhered to. This sequence is derived from the package relations specified in the WITH lists. All package specifications of the packages specified in the WITH lists of a compilation unit must be compiled prior to the compilation unit.

Once a package specification has been modified, the associated package body and all compilation units using this specification must be recompiled.

Applications for Packages

A package should only combine entities which are logically related. In this respect, there are four different areas of applications:

– Set of declarations
  Exports constants, types and variables
  Does not export subprograms

– Set of subprograms
  Does not export constants, types or variables
  Exports subprograms

– Abstract data types
  Exports constants and types
  Exports subprograms
  No status information is stored in the package!

– Automata
  Exports constants and types
  Exports subprograms
  Status information is stored in the package!

These four areas of application are explained in greater detail in the sections that follow.

Sets of Declarations

One of the simplest applications for packages is to combine constants, types and variables. Declarations used by different compilations units may be collected in a package. In this case, any modifications made will only affect one compilation unit.

For example, all machine-dependent system constants of a program can be combined in a package. If the program is then to be ported to another computer, this ensures that these constants can be changed simply and conveniently for the entire program.

```
PACKAGE system_constants;

CONST
   core_size       =   64536;
   printer_width   =      80;
   winchester      =   False;
   terminal_width  =      80;
   terminal_height =      25;

END.



PACKAGE BODY system_constants;
BEGIN
END.
```

On the other hand, it is often useful to combine logically related types. The following package illustrates how definitions might look in conjunction with date information.

```
PACKAGE date_definitions;

TYPE
   name_of_day     = (sunday,   monday,  tuesday, wednesday,
                       thursday, friday,  saturday);
   date_of_day     = 1..31;
   name_of_month   = (january,    february, march,    april,
                       may,        june,     july,     august,
                       september,  october,  november, december);
   date_of_month   = 1..12;
   table_of_days   = ARRAY [date_of_month] of date_of_day;
   date_of_year    = 0 .. Maxint;
   date            = RECORD
                      day   : date_of_day;
                      month : date_of_month;
                      year  : date_of_year;
        END;
CONST
   max_date_of_day = table_of_days
                     (31, 28, 31, 30, 31, 30,
                      31, 31, 30, 31, 30, 31);

END.



PACKAGE BODY date_definitions;
BEGIN
END.
```

Packages should be kept small. The larger the package, the less clear and readable it is. When specifications become hard to read, you should consider subdividing the data structures hierarchically, or choosing a narrower logical context.

Moreover, packages should not be used like common areas, as in the days of FORTRAN. If variables are offered in specifications, each access operation on them is unmonitored. The safe way is to use access functions on status variables in the pakkage body (see also section 17.2.3). This not only makes it possible to monitor the access operation, but the representation of the data can also be changed without affecting the other compilation units. The latter is probably the more important point as it clearly underscores the power of packages as compared to languages of earlier generations.

Set of Subprograms

As with combining constants, types and variables, it is also possible to group different subprograms into a unit by means of a package. For example, the hyperbolic functions can be combined in a package and used as a subprogram:

```
PACKAGE hyperbolic_functions;

FUNCTION sinh (x : Real) : Real;
FUNCTION cosh (x : Real) : Real;
FUNCTION tanh (x : Real) : Real;

END.
```

The subprogram headings given in this specification must be completed in the package body by means of subprogram blocks. The example below illustrates one way of doing this.

```
PACKAGE BODY hyperbolic_functions;

FUNCTION sinh (x : Real) : Real;
BEGIN
   sinh := (Exp (x) + Exp (-x)) / 2.0
END;

FUNCTION cosh (x : Real) : Real;
BEGIN ... END;

FUNCTION tanh (x : Real) : Real;
BEGIN ... END;

BEGIN
END.
```

Abstract Data Type

By "abstract data type" we mean the union of type and operations which are permitted on values of this type. Packages make it possible to define abstract data types. For reasons of clarity, we recommend setting up a separate package for each abstract data type. In order to induce logical abstraction, the type is best declared as a so-called "private type". Admittedly, this is only possible in the case of Pointer types.

By "private pointer type" we mean a type whose type denoter is known externally, but whose domain type is invisible outside the package. In this way, it is only possible to access values of this type via the defined access functions. A private pointer type is defined in the specification as a pointer with a domain type which is not defined until the package body.

Section 11.4 uses the example of a wait queue to show how to implement an abstract data type.

Automatons

An automaton is described by a set of states and state transitions. At any given time the automaton is in exactly one state. When an automaton is implemented by means of a package, its given state is represented by the momentary values of the variables declared in the package body. The state information is inaccessible to a user of the automaton. The automaton can modify its state by performing operations (procedures). With the aid of functions, the user of the automaton can obtain information on its momentary state.

In general, an automaton does not export constants, types or variables. Hence, its appearance resembles a set of subprograms. The major difference lies in the state information stored in the package. In contrast, the subprograms in the package hyperbolic_functions are independent of each other as the result of a function does not depend on other function calls. With an automaton, the result of a subprogram call is dependent on all the preceding calls.

```
PACKAGE integer_stack;

CONST
   full  = 900; { exception for stack overflow }
   empty = 901; { exception when accessing empty stack }

PROCEDURE push (i : Integer);
   { an Integer value is written to the stack }

PROCEDURE pop  (VAR i : Integer);
   { an Integer value is taken from the stack }

FUNCTION  is_empty : Boolean;
   { any more Integer values on the stack? }

PROCEDURE clear;
   { puts the stack in the basic state }

END.
```

The package may, for example, be used to implement a pocket calculator. The entered values are written to the stack. For this purpose, the procedure "push" is offered. With the aid of "pop", values can again be taken from the stack. The two constants "empty" and "full" define the possible exception situations of the automaton. The procedure "clear" puts the automaton in the basic state.

```
PACKAGE BODY integer_stack;

CONST
   stack_size  = 100;

TYPE
   stack_array = ARRAY [1..stacksize] OF Integer;

VAR
   stack      : stack_array;
   stack index: 0..stack_size;

PROCEDURE push (i : Integer);
   { a value is written to the stack }
BEGIN
   IF stack_index = stack_size THEN
      Raise (full);
   stack_index         := stack_index + 1;
   stack [stack_index] := i;
END { push };

PROCEDURE pop  (VAR i : Integer);
   { a value is taken from the stack }
BEGIN
   IF is_empty THEN
      Raise (empty);
   i           := stack [stack_index];
   stack_index := stack_index - 1;
END { pop };

FUNCTION  is_empty : Boolean;
   { any more values on the stack ? }
BEGIN
   is empty := stack_index = 0;
END { is_empty };

PROCEDURE clear;
   { puts the stack in the basic state }
BEGIN
   stack_index := 0
END { clear };

BEGIN
   clear;
END { integer_stack }.
```

The entities stack and stack_index form the internal state information. Since they are declared globally in the package, they exist during the entire time the program is being executed. These data items can only be accessed by the visible procedures push, pop and clear and the function is_empty.

The stack was implemented by means of an array. This limits the number of values which can be written to the stack. Should the representation of the stack be changed (e.g. into a chained list) for an expanded implementation, this will have no effect on compilation units which use the package "stack".

Exception Handling Concept

Exceptions are special cases of an algorithm which do not occur when the algorithm runs normally. Taking them into account does not help us to understand the algorithm; on the contrary, it only makes the algorithm more difficult to understand. For example, when calculating with integers, the integer values may be too large to be representable in the computer (e.g. they may no longer lie within the corresponding Integer type). To take this special case (arithmetic overflow) into account in a numeric algorithm, before any formula is evaluated it would have to be preceded by a complicated interrogation with the values of the operands used in it to see whether the intermediate or final results might turn out to be too large. This interrogation would doubtless make the numeric algorithm completely unintelligible. For this reason, it is better to write the actual algorithm first without taking these special cases (exceptions) into account, and then provide measures for handling the exceptions in a second part (the exception handling part or EXCEPTION part). The measure in the case of an arithmetic overflow, for example, would be to issue a message indicating that the numeric values became too large and that the computation could therefore not be carried out with the numeric representation used in the computer. As a consequence, after receiving this message we would abort the program since further processing would be pointless.

In contrast, the special cases of an algorithm which might even occur when the algorithm runs normally, or which contribute to our understanding of the algorithm when taken into account, are not exceptions in the above sense of the term. It is therefore thoroughly possible for one and the same event to be considered an exception in one algorithm but a normal case in another. Consider, for example, what happens when an end-of-tape mark is reached when reading input data. If a file is to be read and processed one component at a time, then reaching the end-of-file is an expected event that causes the algorithm to terminate normally.

Example

```
PROGRAM process (data);
TYPE
   t = ... ;
VAR
   data: FILE OF t;
   elem: t;
BEGIN
   WHILE NOT Eof (data) DO BEGIN
      Read (data, elem);
      { process }
      END;
END.
```

If, however, three numbers are to be read from a textfile, then reaching the end-of-file prematurely may be considered an exception in the above sense of the term.

Example

```
PROGRAM example (file, Output);

VAR file: Text;
    num1,
    num2,
    num3: Integer;

BEGIN
   Reset (file);
   Read  (file, num1, num2, num3);
   ...
EXCEPTION
   IF Error_Number = Eof_Error THEN
      Writeln ('Error in file')
   ELSE
      Writeln ('Other error')
END.
```

In Pascal-XT, exceptions are coded by means of Integer-type numbers. Negative numbers are reserved for predefined exceptions; positive numbers may be used for user-defined exceptions. Constants have been declared for the predefined exceptions (see section 14.1).

Programmed exception handling

When an exception (error) occurs, it makes little sense simply to continue with program execution and pretend as if nothing had happened. Instead, depending on the situation involved, the following measures may profitably be followed:

– Abort the entire program with an error message and, if applicable, start any necessary termination routine to leave data collections on files in a consistent state.

– Abort the subprogram where the exception occurred. Here, too, it may be necessary beforehand to perform termination handling to leave global data structures in a consistent state so that it is possible and meaningful to resume the program behind the call of the aborted subprogram.

– Execute an alternative algorithm which accomplishes, with more complexity, the same thing as the algorithm interrupted by the occurrence of the exception. In the case of memory overflow Memory_Error), for example, further data might be written to a file instead of being stored in main memory.

– Repeat the operation which led to the exception situation. Repetition is obviously meaningful only if we can assume that the same exception situation will not occur the second time. If, for example, an attempt to open a file (with Reset or Rewrite) causes an Open_Error because the the file has just been opened by another program, it makes sense to try opening the file a second time following an interval because it may be available (closed) in the meantime. In this case, we can assume that the cause of the error will rectify itself after a while. In general, however, there is no point in repeating the error-causing operation without removing the cause of the error beforehand.

For programming all of these possible responses, Pascal-XT has provided the concept of exception handling. An exception handler can be specified for a subprogram or a compound statement (see chapter 14). If an exception occurs, program execution continues from the dynamically most recent exception handler. To do this, a search begins within the currently active subprogram or main program for the innermost compound statement with exception handler containing the statement that caused the exception. If there is no such compound statement in the currently active subprogram, the search will continue in the same way in the dynamic predecessor which called this subprogram. There, a search takes place for the innermost compound statement with exception handler containing the subprogram call. The search continues until either an exception handler is located or, if none is found, the program is aborted with an error message from the runtime system.

Now let’s consider these possible responses to an exception in detail:

- **Aborting a program or subprogram**

The exception handler is specified in the block of the subprogram or main program. It has the task of performing termination handling. Then the entire program (subprogram) is aborted.

Example

```
PROGRAM example (old_data, new_data, transaction, Output);
VAR
   old_data,
   new_data,
   transaction : Text;

BEGIN
   Reset   (old_data);
   Reset   (transaction);
   Rewrite (new_data);
   ...
   { execute the transactions }
   ...
EXCEPTION
   { new_data may be incomplete, i.e. delete }
   Rewrite (new_data);
   Writeln (Output, 'Program aborted');
END
```

- **Executing an alternative algorithm**

The procedure below is designed to search for an opening move in a library. If this library cannot be opened, processing continues with the normal heuristic search for a suitable opening move.

Example

```
PROCEDURE openmove;
BEGIN
   Reset (openlib);
   search_move;     { exception 13 if nonexistent }
EXCEPTION
   IF (Error_Number = Open Error) OR
      (Error_Number = 13)         THEN
      own_move
   ELSE
      Raise (0);
END
```

This example also illustrates how to propagate all exceptions which are unexpected or cannot be handled. This is done by calling Raise for all unexpected error numbers (see section 15.11).

- **Repeating the error-causing operation**

The operation to be repeated in case of error (exception) is packed in a compound statement with exception handler. This compound statement is in turn the body of the repeat loop. The following example illustrates how to attempt to open a file three times before aborting the program with an error message. Once the file has been successfully opened, the loop is abandoned with the EXIT statement (see section 10.1.5).

Example

```
FOR i := 1 TO 3 DO
   BEGIN
      Reset (file);
      EXIT;
   EXCEPTION
      IF (i = 3) OR (Error_Number <> Open_Error) THEN
         Raise (13);
   END;
```

This example illustrates how to create user-defined exceptions: namely, by calling Raise (see section 15.11) with an error number greater than zero. In the example, the exception with the user-defined error number 13 will be created with the occurrence of an error other than Open_Error or with the third repetition. Since Raise (13) is located within the exception handler, the exception is immediately propagated to a superordinate exception handler (see section 14.3). Raise (13) also abandons the FOR loop.

However, it should also be mentioned in this context that Raise must not be specified as a special form of GOTO. Raise should only be used to create genuine exception situations! Seen in this light, the Raise (13) in our example is justified.

Cross-references

Compound statement:	10.2, 14.2
Exception:	14
EXCEPTION part:	14.2
Exception handling:	14.3
Error Number, Raise:	15.11

Input/Output

Experience has shown that Pascal novices usually have problems with input/output. For this reason, the concepts are repeated here and the problematical aspects are dealt with in somewhat greater detail. An exact description of the required procedures and functions can be found in section 15.1.

The Pascal standard only discusses file processing on the programming level. In view of the vast selection of operating systems, data recording techniques and so forth available, the external representation of a file (input/output devices and data media are both regarded as files in this context) cannot be made subject to programming language standardization. To distinguish between external files and their internal (programmable) representations, the terms "logical file" and "physical file" have been generally adopted.

Implementation-defined characteristic

Pascal only describes the logical effects of file operations. The physical activities and the time of their execution are implementation-defined.

Throughout this chapter, logical files are represented in the following form:

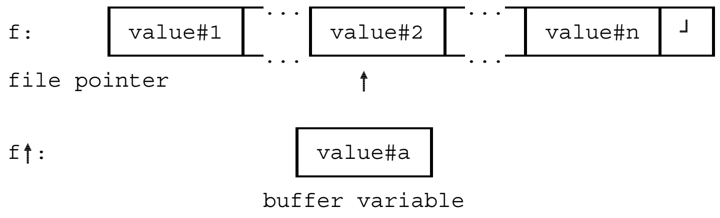


Fig. 19-1 Representation of a logical file

Let f be a variable of type FILE OF t. Then values of f are a sequence of values of the component type t which can be read- or write-accessed. The current position, also referred to as file pointer (represented by ↑), describes the current read- or write-positions. For each FILE variable there exists implicitly a buffer variable f↑ of type t which can accommodate a value from the sequence. Values in the file can be accessed only via the buffer variable.

In all the figures below, the end of the sequence is represented by the character ┘. This character is used for illustrative purposes only and does not imply any particular implementations.

Data transport between the buffer variable and the physical file is described by means of the required procedures Get, Read, Put, Write, Reset and Rewrite.

Standard Pascal only recognizes sequential files. Pascal-XT implementations may additionally offer predefined packages in order to support, for example, direct access files or index-sequential files.

The end of a file can be determined with the required function Eof. This function yields the Boolean value True if the end of file f has been reached; otherwise it yields the Boolean value False. The application of this function is only meaningful for reading opened files, since with writing one is always at the end of file anyway. Once the end of the file has been reached, any further attempt to read values will cause a runtime error.

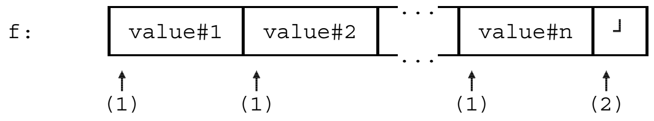


Fig. 19-2 Applying the required function Eof

If while reading you find yourself at the positions marked with (1), then Eof(f) is False. At the positions marked with (2), Eof is True.

Files in Pascal fall into two main categories:

- **Local files**

  A file is called a local file if the identifier of the FILE variable is not specified as a program parameter in a main program. The assignment to a physical file during program execution is performed implicitly by an implementation-dependent mechanism. In Pascal-XT, a physical file may also be assigned to a local file by means of Assignfile.

The lifetime of a local file is linked to the block where it was declared as a variable:
- Local files declared in a main program or package exist for the entire duration of program execution.
- Local files declared in a subprogram are created when the subprogram is called, and destroyed when it is terminated.
- Identified variables and components of identified variables may be of a FILE type. Their lifetime then corresponds to that of these variables (created with New).

The physical file assigned to a local file likewise ceases to exist when the block is terminated.

• **External files**

External files are always declared directly in the main program block or in a package block. They are reported as external variables (i.e. variables existing independently of the program) by specifying their variable identifier in the program parameter list. These variable identifiers are assigned physical files at program execution time. External files must be neither components of variables nor identified variables.

To process a file you have to do the following:

1) Declare a FILE variable of the desired FILE type
   - as a local or global variable, or
   - as an identified variable or a component thereof.

2) Establish a link to an external file when using external files.

3) Specify the required textfiles Input and Output in the program parameter list if they are used in the program. They are assigned the data display terminal by default.

Cross-references

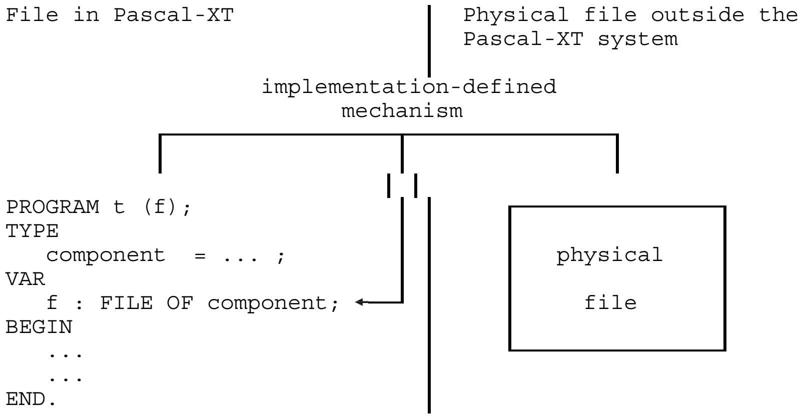
FILE type:	6.3.5
Textfile:	6.3.5.2
Buffer variable:	9.6.5
Input/Output:	11.5
Block:	12.1
Program execution:	13.3
Required subprograms:	15.1

Assigning a Physical File

Assigning a physical file to a FILE variable can be done in either of two ways: by specifying the variable identifier in the program parameter list, or by means of the required procedure Assignfile.

- **Specification in the program parameter list**

Assigning a physical file to a FILE variable specified as a program parameter is done by means of an implementation-defined mechanism.

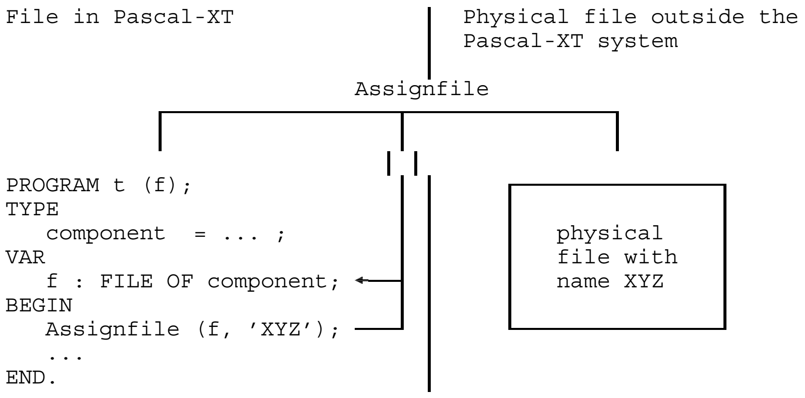

File in Pascal-XT
Physical file outside the Pascal-XT system
implementation-defined mechanism
physical file

```
PROGRAM t (f);
TYPE
   component  = ... ;
VAR
   f : FILE OF component;
BEGIN
   ...
   ...
END.
```

Fig. 19-3a Assignment via an implementation-defined mechanism

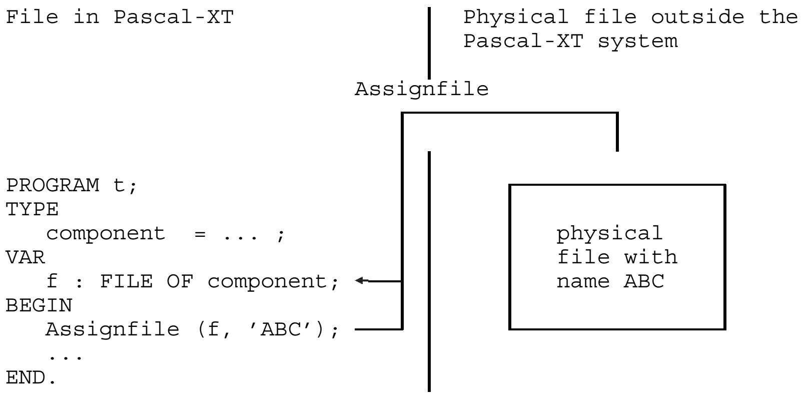
- **Assignment via Assignfile**

The required procedure Assignfile can be used to assign a physical file either to a local or to an external file. The way the physical file is described in the second parameter of Assignfile is implementation-defined.


File in Pascal-XT
Physical file outside the Pascal-XT system
Assignfile
physical file with name XYZ

```
PROGRAM t (f);
TYPE
   component  = ... ;
VAR
   f : FILE OF component;
BEGIN
   Assignfile (f, 'XYZ');
   ...
END.
```

Fig. 19-3b Assigning an external file with Assignfile


File in Pascal-XT
Physical file outside the Pascal-XT system
Assignfile
physical file with name ABC

```
PROGRAM t;
TYPE
   component  = ... ;
VAR
   f : FILE OF component;
BEGIN
   Assignfile (f, 'ABC');
   ...
END.
```

Fig. 19-3c Assigning a local file with Assignfile

Opening Files for Reading or Writing

Before files can be processed for reading or writing they must be opened with Reset or Rewrite, respectively

Reset (f)

Reset (f) opens file f for reading and moves the first component of the physical file to the buffer variable. The file pointer is positioned to the first component in the file. If the file is empty, Eof (f) is True and the buffer variable f↑ is undefined.

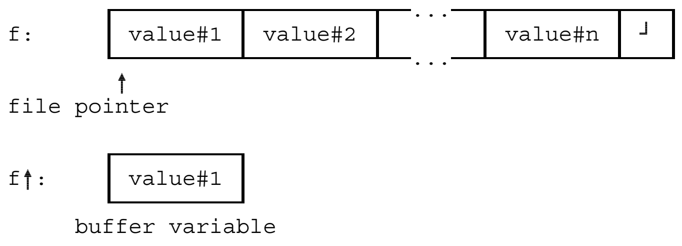


Fig. 19-4 Opening a non-empty file for reading

Rewrite (f)

Rewrite opens file f for writing. Following Rewrite (f) the file is empty, and any earlier file contents are lost. The buffer variable f↑ does not have a defined value, and Eof (f) is True.

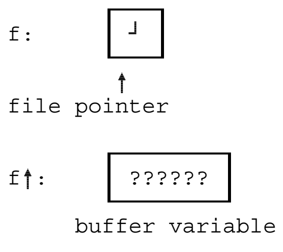


Fig. 19-5 Opening a file for writing

The Read/Write Procedures Get and Put

The required procedures Get and Put are basic functions for moving values from the physical file to the buffer variable or vice versa.

Get (f)

Get (f) shifts the file pointer one position further. If there is another element in the file, it is moved to the buffer variable f↑ and Eof (f) yields the value False. If not, the buffer variable f↑ is undefined and Eof (f) is True.

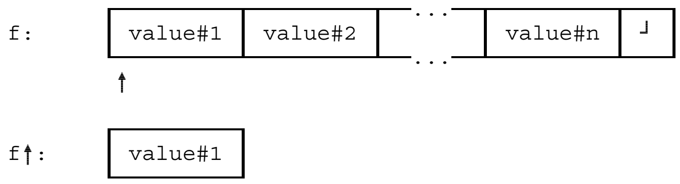


Fig. 19-6a Status after Reset (f)

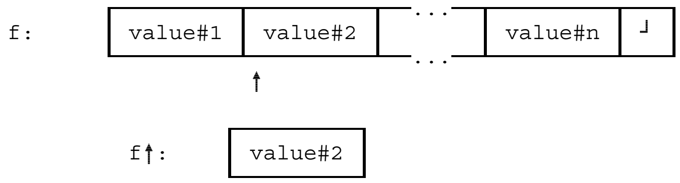


Fig. 19-6b Status after the first Get (f)

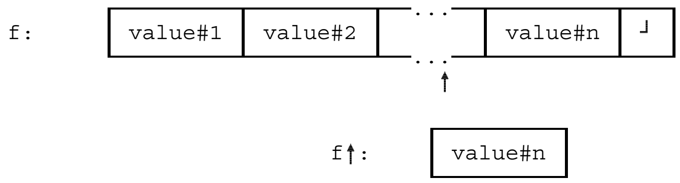


Fig. 19-6c Status after the (n-1)-th Get (f)

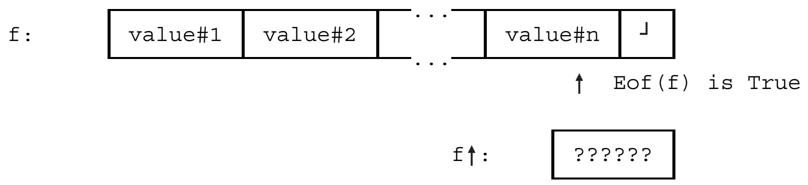


Fig. 19-6d Status after the n-th Get (f) (end of file)

Put (f)

Put (f) appends the current value of the buffer variable f↑ to the end of the file opened with Rewrite. Thus, before Put is called, the buffer variable must be assigned a value. After Put is called, the contents of the buffer variable are undefined.

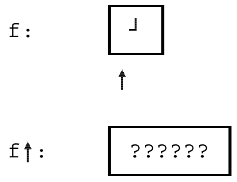


Fig. 19-7a Status after Rewrite (f)

```
f↑ := value#1;
Put (f);
```

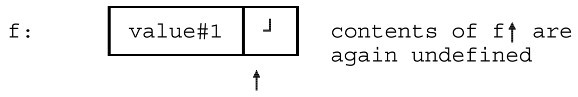


Fig. 19-7b Status after Put (f)

```
f↑ := value#2;
Put (f);
```

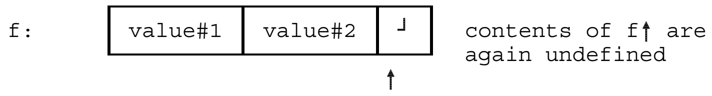


Fig. 19-7c Status after second Put (f)

The Read/Write Procedures "Read" and "Write"

Since we primarily want to move data from program variables to files or vice versa, the file access via the buffer variable is annoying. For this reason, the required procedures Read and Write have been defined as a means of abbreviating read and write operations.

Read (f, v1, .., vn)

The Read parameter list can be used to specify any number of Read parameters vi (but at least one). A Read statement of this sort is then regarded as a sequence of individual Read statements, each with only one Read parameter. Read (f, v) is short for the following statement sequence:

```
BEGIN v := f↑; Get (f); END
```

In other words, Read moves the current component from the buffer variable to the variable, and the next component from the file to the buffer variable. The following sequences of instructions illustrate the differences when programming loops, depending on whether Get or Read is used.

```
Reset (f);                           | Reset (f);
WHILE NOT Eof (f) DO BEGIN           | WHILE NOT Eof (f) DO BEGIN
   { process f↑ }                    |    Read (f, v);
   Get (f);                          |    { process v }
   END;                              |    END;
```

The figures below illustrate different statuses when reading components from a file.

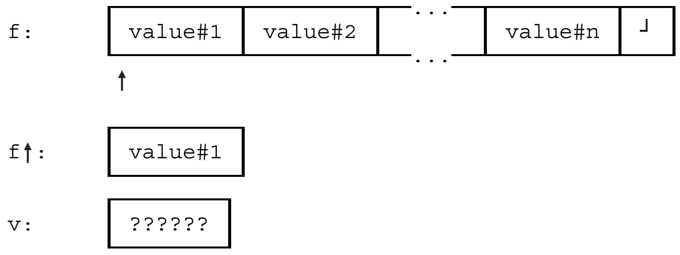


Fig. 19-8a Status after Reset (f)

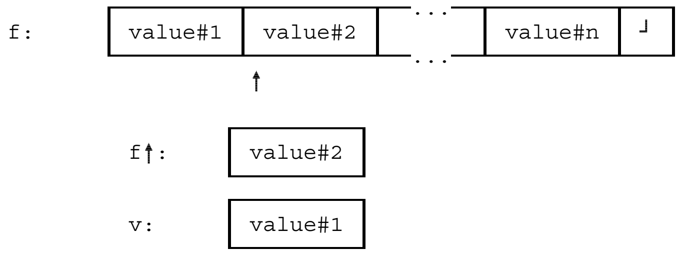


Fig. 19-8b Status after the first Read (f,v)

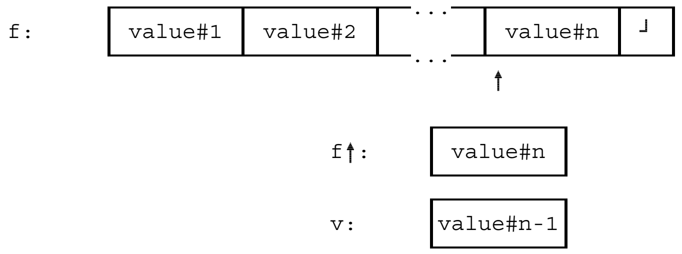


Fig. 19-8c Status after the (n-1)-th Read (f,v)

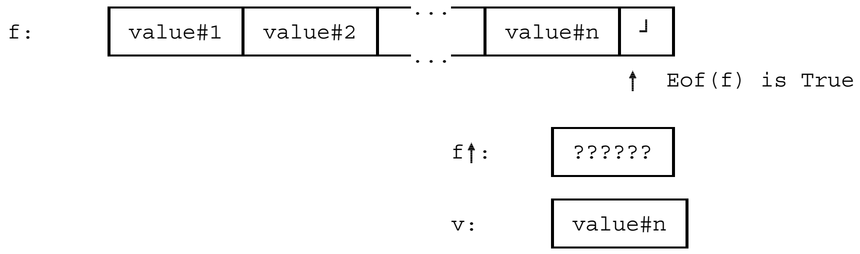


Fig. 19-8d Status after the n-th Read (f,v) (end of file)

Write (f, a1, ..., an)

The Write parameter list can be used to specify any number of parameters ai (but at least one). A Write statement of this sort is then regarded as a sequence of individual Write statements, each with only one parameter. Each Write (f, a) is short for the following statement sequence:

```
BEGIN f↑ := a; Put (f); END
```

If the FILE variable f is not of the required type Text, the expressions ai must be assignment-compatible with the component type of the FILE type. If f is a textfile, the ai's are referred to as write parameters, and may have the types described in section 19.5.

The figures below illustrate different statuses when writing components to a file.

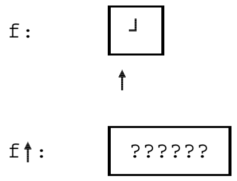


Fig. 19-9a Status after Rewrite (f)

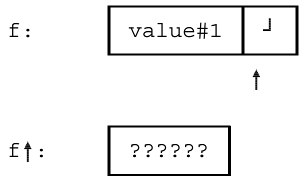


Fig. 19-9b Status after Write(f, value1)

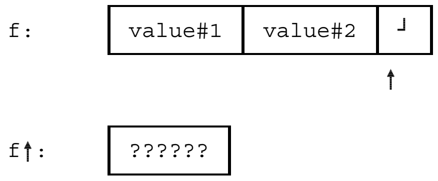


Fig. 19-9c Status after Write(f, value2)

The sample programs below are equivalent examples for copying a file with Real numbers. If Get and Put are employed, a component is copied immediately from the buffer variable of the input file to the buffer variable of the output file. If Read is chosen, copying takes place with a detour via an auxiliary variable.

```
PROGRAM copy(f,g);

VAR
   f, g: FILE OF Real;


BEGIN
   Reset(f);
   Rewrite(g);
   WHILE NOT Eof(f) DO BEGIN
      g↑ := f↑;
      Put(g); Get(f);
      END
END.

Copying a file with
Get and Put
```

```
PROGRAM copy(f,g);

VAR
   f, g: FILE OF Real;
   r   : Real;

BEGIN
   Reset(f);
   Rewrite(g);
   WHILE NOT Eof(f) DO BEGIN
      Read(f,r);
      Write(g,r);
      END
END.

Copying a file with
Read and Write
```

Textfiles

Files of the required type Text (called "textfiles" for short) have an additional property: they describe a sequence of lines, with each line consisting of a sequence of Char-type characters. Each line ends with a special character, the end-of-line (EOL) component. This component is treated like a blank within a Pascal program. A textfile may only contain complete lines, as illustrated in Fig. 19-10.

Textfiles must not be confused with files of type FILE OF Char, which likewise consist of a sequence of characters but do not have a line structure. The subprograms additionally defined for textfiles cannot be applied to them. For both files, the buffer variable is of type Char.

Figures 19-10 and 19-11 again illustrate the difference between these two types of file.

In all the examples below, the end-of-line component is represented by the character "•" and the end-of-file component by '┘'. Blanks are represented by '_' for better visibility.

```
VAR f1: Text;
    f2: FILE OF Char;
```

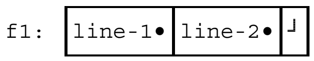


Fig. 19-10 Structure of a textfile

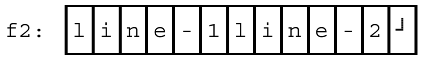


Fig. 19-11 Structure of a file of type FILE OF Char

There are four additional required subprograms for textfiles:

- `Readln`
- `Writeln`
- `Eoln`
- `Page`

Readln and Writeln are used for handling the end-of-line component. Writeln creates this component, and Readln can be used to skip it. Eoln queries whether the end of line has occurred. It yields the value True when the file pointer points to the end-of-line component (the buffer variable then contains a blank); otherwise, the value is False. Fig. 19-12 shows the value of Eoln for different positions of the file pointer.

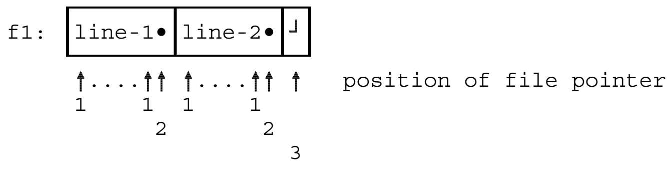


Fig. 19-12 Use of the required function Eoln

If, when reading, you are at the positions marked with (1), then Eoln(f1) is False. At the positions marked with (2), Eoln(f1) is True. At position (3), Eoln(f1) must not be called because Eof(f1) is already True.

The Page procedure is used for formatting pages of textfiles which are output to printer. Text written to the file after a Page call is put on a new page when printed out.

Input and Output are required FILE variables of type Text with special properties. They must not be declared in a program. As soon as they are specified in the program parameter list, they become visible and are automatically opened for reading or writing when the program is started. Both files are usually assigned the data display terminal.

Unlike non-textfiles, the parameters of the required procedures Read, Readln, Write and Writeln can have the following types:

– the Char type
– Integer and Real types
– fixed and variable string types
– the Boolean type (for Write or Writeln).

When numbers are output with Write or Writeln, they are automatically converted from internal representation into printable characters. When numbers are read with Read or Readln, conversion takes place in the reverse order.

With Boolean values, the character string ’TRUE’ or ’FALSE’ is output (the orthography is implementation-defined).

The FILE value may be omitted in subprogram calls for textfiles. In this case:

– Readln, Read, Eoln and Eof refer to the required textfile Input, and
– Writeln, Write and Page refer to the required textfile Output.

Reading from a Textfile

Read (f, v1, ..., vn)
Readln (f, v1, ..., vn)

The Read procedure reads values from the current line of textfile f and moves them to the Read parameters v1 to vn. Readln functions in the same way as Read, but has the additional property that *after* the Read parameters v1 to vn are read, the remaining characters of the current line, if any, are skipped and the first character of the next line is put in the buffer variable.

Each Read parameter vi must be a variable access with one of the types Char, Integer, Real, or a fixed or variable string type.

The figures below illustrate the situations that arise when reading values from a textfile (other than Input). Blanks are represented as '_' for better visibility.

Note

The peculiarities of Readln in conjunction with input from interactive terminals are described in section 19.5.2.

```
VAR f:  Text;
    r:  Real;
    c1,
    c2: Char;
```

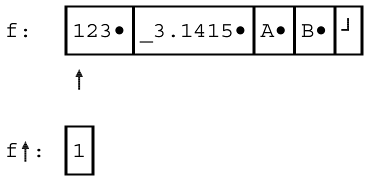


Fig. 19-13a Status after Reset (f)

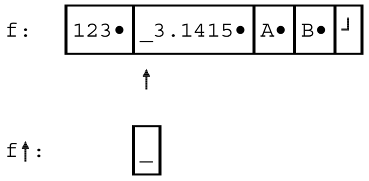


Fig. 19-13b Status after Readln (f)

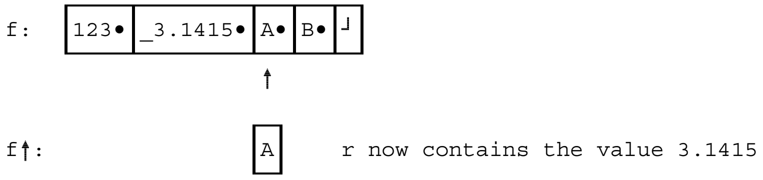


Fig. 19-13c Status after Readln (f, r)

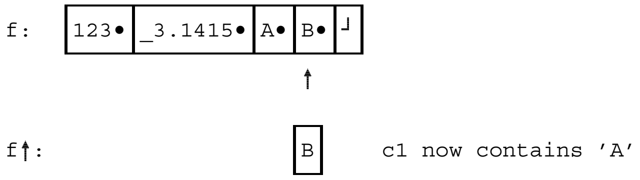


Fig. 19-13d Status after Readln (f, c1)

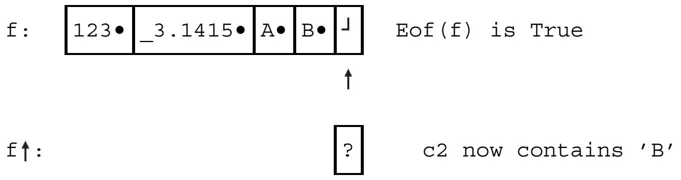


Fig. 19-13e Status after Readln (f, c2)

Reading characters

If v is a variable of type Char or a subrange of Char, then Read (f, v) is defined by

```
     BEGIN v := f↑; Get (f) END
```

The variable v contains the blank '_' if, prior to the Read (f, v) call, Eoln (f) is True.

```
VAR f: Text;
    c: Char;
```

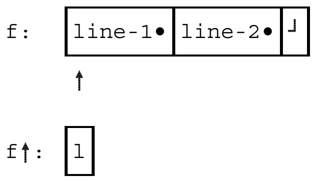


Fig. 19-14a Status after Reset (f)

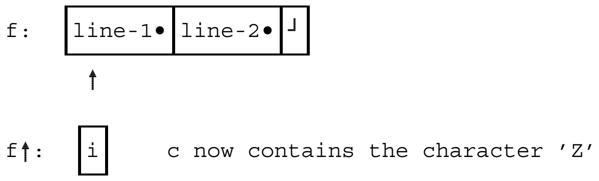


Fig. 19-14b Status after Read (f, c)

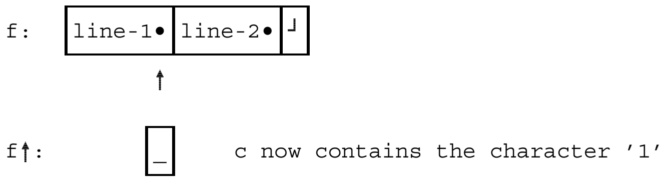


Fig. 19-14c Status after five further Read (f, c) calls

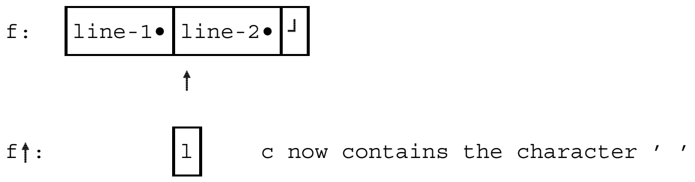


Fig. 19-14d Status after the next Read (f, c)

Reading Integer numbers

If v is a variable of type Integer or a subrange thereof, a sequence of characters is read. Leading blanks and end-of-line components are skipped. Immediately in front of the number there may be a sign. The Read operation is terminated when an entered character can no longer be part of an Integer number. This character is then located in f↑.

The input digit sequence must correspond to an Integer number in decimal notation in accordance with section 3.5. The value corresponding to this digit sequence is assigned to variable v.

```
VAR f: Text;
    i: Integer;
```

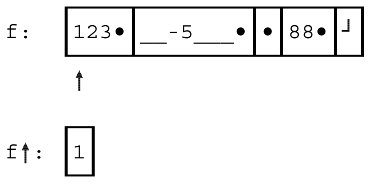


Fig. 19-15a Status after Reset (f)

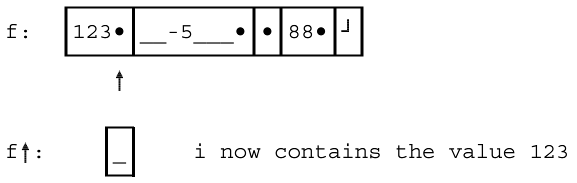


Fig. 19-15b Status after Read (f, i)

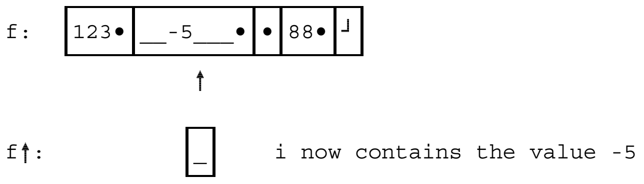


Fig. 19-15c Status after the next Read (f, i)

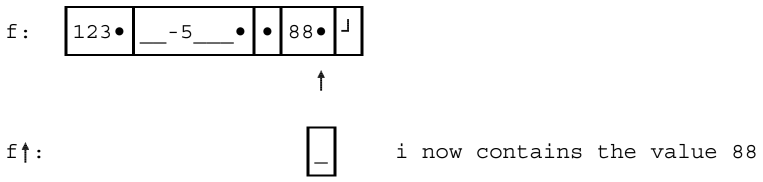


Fig. 19-15d Status after the next Read (f, i)

Reading Real numbers

If v is a Real-type variable, a sequence of characters is read. Leading blanks and end-of-line components are skipped. Immediately in front of the number there may be a sign. The Read operation is terminated when an entered character can no longer be part of a Real number (in accordance with section 3.5). This character is then located in f↑. The value of the read number is then moved to v.

```
VAR f: Text;
    r: Real;
```

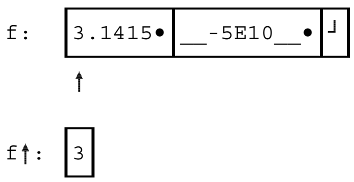
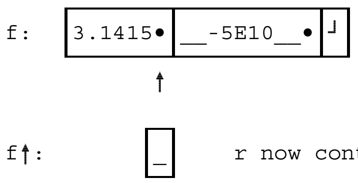
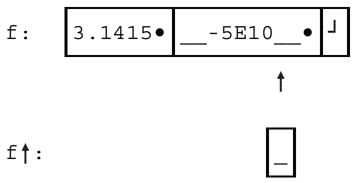


Fig. 19-16a Status after Reset (f)



r now contains the value 3.1415

Fig. 19-16b Status after Read (f, r)



r now contains the value $-5 \cdot 10^{10}$

Fig. 19-16c Status after the next Read (f, r)

Read parameters of a variable string type

If v is a variable of a variable string type (String), all characters up to the end of the current line are read and moved to v. After the Read operation, Eoln (f) is True and f↑ contains the end-of-line component.

```
 VAR f: Text;
     s: String;
```

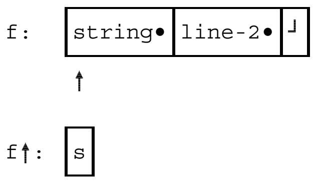


Fig. 19-17a Status after Reset (f)

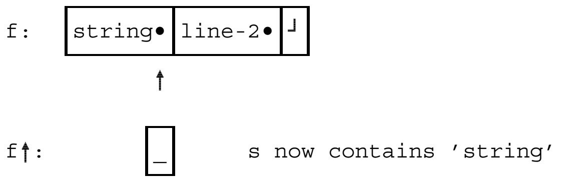


Fig. 19-17b Status after Read (f, s)

A Readln (f) call is now imperative since otherwise, with a subsequent Read (f, s), nothing will be read and the file pointer will continue to point to the end-of-line component.

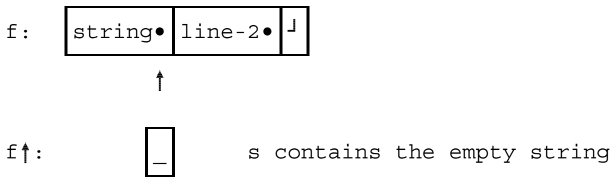


Fig. 19-17c Status after Read (f, s)

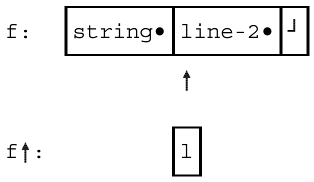


Fig. 19-17d Status after Readln (f)

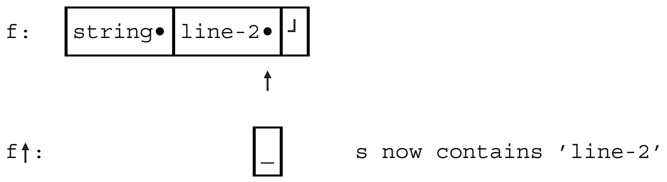


Fig. 19-17e Status after Read (f, s)

Read parameters of a fixed string type

If v is a variable of a fixed string type (PACKED ARRAY [1..n] OF Char), the Read operation is terminated when either the number of characters entered is equal to n, or the end-of-line is reached. In the latter case, the remaining characters of v and padded with blanks.

```
 VAR f: Text;
     a: PACKED ARRAY [1..6] OF Char;
```

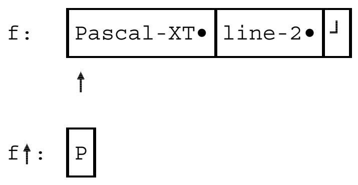


Fig. 19-18a Status after Reset (f)

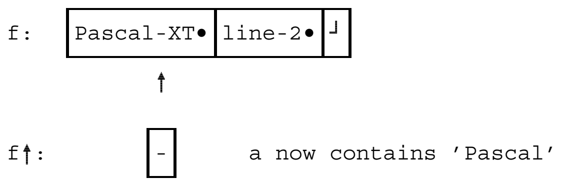


Fig. 19-18b Status after Read (f, a)

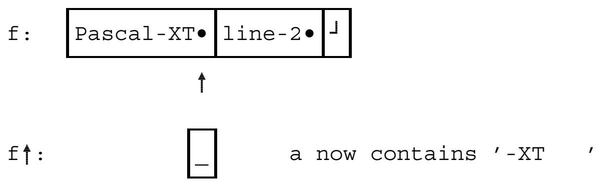


Fig. 19-18c Status after Read (f, a)

A Readln (f) call is now imperative since otherwise, with a subsequent Read (f, a), nothing will be read and the file pointer will continue to point to the end-of-line component.

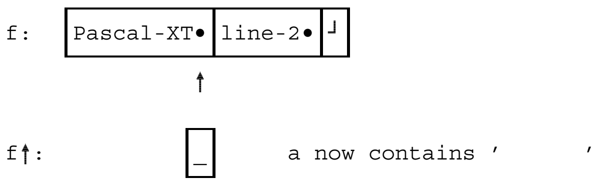


Fig. 19-18d Status after Read (f, a)

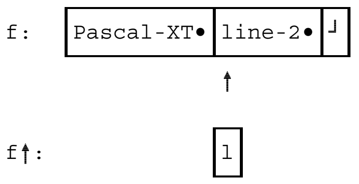


Fig. 19-18e Status after Readln (f)

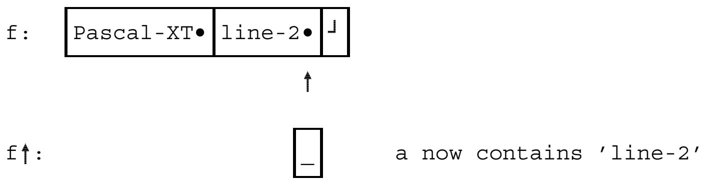


Fig. 19-18f Status after Read (f, a)

Reading from the Terminal

When a program is started the file Input is automatically opened for reading. This means that a Reset (Input) is performed implicitly. However, Reset not only causes the file to be opened for reading, but loads the first component of the file into the buffer variable as well. Since the file Input is assigned to the terminal by default, this would lead to an input prompt. To solve this, Pascal-XT has introduced a virtual line 0 containing solely an end-of-line component. Following the implicit Reset (Input), Eoln(Input) is True and Input↑ contains a blank.

To read a character or character string v from a new line, we recommend the following sequence of calls:

```
Readln (Input);
Read   (Input, v);
```

When an Integer or Real number is read, all leading blanks and end-ofline components are skipped. The preceding Readln may therefore be omitted.

Another problem when reading from terminal is the behavior of Readln. Readln, of course, skips the remaining characters in the current line and positions control to the first character in the next line. On the terminal, however, there is no next line as yet; it must first be entered by the user. This means that each Readln, and even each read operation on the end-of-line component with Read, causes the user to be given an input prompt. A statement in the form

```
Readln (Input, v1,..., vn)
```

causes the values for variables v1 to vn to be read in first, followed by a prompt for a new line.

Writing to a Textfile

Write (f, p1,..., pn)
Writeln (f, p1,..., pn)

The Write procedure writes values (Write parameters) p1,...,pn to the current line of the textfile f. Writeln works like Write except that it also causes the current line to terminate with an end-of-line component and control to be positioned to the beginning of a new line.

In Pascal-XT, output caused by Write is first sent to an internal line buffer. Only after Writeln does output take place to the file or (if Output is specified) to the terminal screen.

The individual Write parameters may have the form

a or a:a1 or a:a1:a2 .

a must be an expression of type Char, Integer, Real, Boolean or String (fixed or variable). At output time, Integer- or Real-type values are converted to decimal representation. Boolean-type values are output as a string ('True' or 'False', where the orthography is implementationdefined).

The expression a1 determines the total length in which the value is output. If a1 is greater than required to represent the value, the value is stored in the field with right-justification. If a1 is less than the required length, the following applies:

- For Integer- or Real-type values, the required number of characters is output.
- Strings of a character string type are output in the length a1; the remainder is truncated.
- Boolean-type values are output like values of a character string type.

If a1 is omitted, an implementation-defined number of characters is output for Integer, Real and Boolean values. Values of type Char are output with a length of 1, character strings in their actual length.

The expression a2 can only be specified for Real-type values. When specified, it causes Real numbers to be output in fixed-point form with a2 digits after the decimal point. If a2 is greater than required for maximum-precision output, zeros will be appended to the decimal digits. If it is less than required, the decimal digits will be rounded.

The figures below illustrate the output formats for Integer and Real values. We have omitted the formats for the other types because of their simplicity.

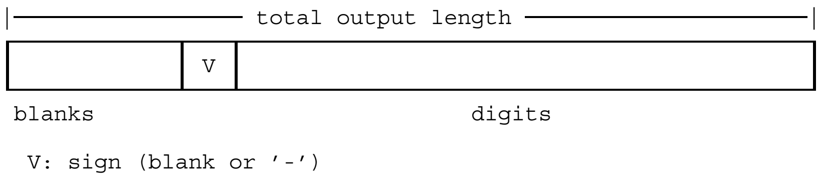


Fig. 19-19 Output format for Integer values

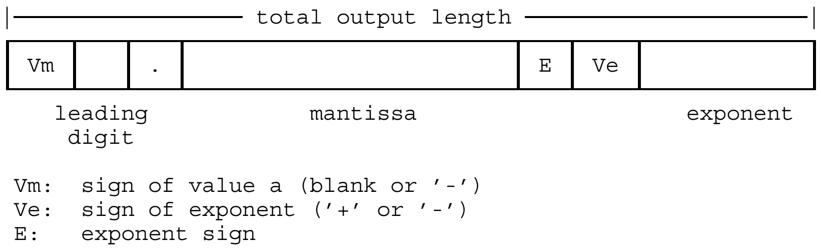


Fig. 19-20 floating-point representation of Real values

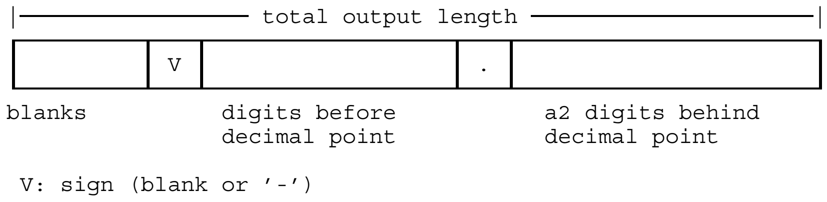


Fig. 19-21 Fixed-point representation of Real values

Dynamic Data and Memory Allocation

Pascal distinguishes between two types of variables:

- static (declared) variables and
- dynamic (identified) variables.

Static variables are declared in a variable declaration such as "VAR a: person;". The compiler reserves memory space for static variables in the block where they were declared. They are created when the block is entered and destroyed when the block is terminated.

Dynamic variables (henceforth called identified variables) are not created until runtime. They are allocated memory space in their own memory area, called the "heap". To process them, one declares a Pointer variable with the variable’s type as domain type. With this form of declaration, the Pointer variables are strictly bound to their domain type and can only point to (identify) dynamic variables (identified variables) of the same type.

Values which can be assumed by Pointer variables are called "identifying values". The possible values of a Pointer type include the NIL value (a defined, empty identifying value) and the set of identifying values that point to an identified variable of the domain type. Identifying values arise exclusively by applying the required procedure New to Pointer variables. Let p be a Pointer variable with domain type t. By applying New (p), an identified variable of type t is created, and p is assigned the identifying value pointing to this variable.

Dynamic memory areas are usually generated and managed by means of required procedures. Standard Pascal provides the New procedure to create memory space for identified variable, and the Dispose procedure to release that memory space. Besides these required procedures, Pascal-XT has additional variants to set up Pointer variables on the heap in abbreviated form with particular domain types (see section 15.2). This saves memory space. However, these abbreviated identified variables can only be accessed one component at a time. As an additional extension, the procedures Mark and Release have been provided. They simplify dynamic memory allocation since a Release call releases (deallocates) all variables created with New after the associated Mark call.

A Pointer variable p only has a defined value following a New (p) call. Referencing this variable before the New (p) call will lead to a runtime error with unpredictable effects. The identified variable p↑, once created, is totally undefined.

The identified variable p↑ is accessed by dereferencing the Pointer variable p. As an extension to the Standard, Pascal-XT has introduced the concept of a dereferenced object to enable dereferencing not only of Pointer variables but also of function calls whose result type is a Pointer type (see section 9.6.4).

An identified variable can be made inaccessible not only by calling one of the two procedures Dispose or Release, but also by assigning another identifying value to the Pointer variable p.

To process two or more variables of the same domain type simultaneously, you can

– declare as many Pointer variables as you need to match the required number of identified variables, or

– chain the identified variables to each other.

The latter option is not only more flexible and more elegant, it also constitutes the main application of pointers altogether, namely to set up dynamic data structures (e.g. chained lists). Chaining is performed by declaring a component of a Pointer type in the associated domain type (which must be a RECORD in this case). This can be done e.g. as follows:

```
TYPE
   list    = ↑element;
   element = RECORD
                next: list;
                ...
             END;
```

Once again let’s summarize the essential characteristics of dynamic data structures and their practical significance:

- Memory space is occupied only as long as it is required. Above all, this can be determined by the programmer himself.
- Any data structure can be set up.

A chained list (Example 1) and a binary tree (Example 2) illustrate the possible applications of dynamic data structures.

Cross-references

Pointer types:	6.4
Variables:	7
Objects:	9.6
Dereferencing:	9.6.4

Example 1: Chained list

Our task is to process a list of names and telephone numbers, putting the names in alphabetical order (character string comparison). One data structure typical for processing such lists is defined as follows:

```
TYPE
   string17  = String[17];
   plist     = ↑listtype;
   listtype  = RECORD
                  name,
                  telephone_no : string17;
                  successor    : plist;
               END;
VAR
   list  : plist;
```

The component "successor" is to identify the list element whose name immediately follows alphabetically. A list defined in this way is called a singly chained list. Similarly, we can insert a RECORD field for reverse chaining, e.g. "predecessor : plist", where "predecessor" then identifies the list element, the value of whose "name" comes alphabetically before the current element. A list of this sort would then be doubly chained.

In our example, the Pointer type "plist" must be defined before "listtype" because this Pointer type is required for the field "successor". This situation is typical for recursive data structures. While identifiers in Pascal otherwise have to be defined before they are applied, an exception to this rule arises in the case of Pointer types, namely, for the expressed purpose of writing recursive data structures. A domain type can be used in a declaration part to define a Pointer type before this domain type is itself defined (in the same declaration part).

The variable "list" must be initialized with "list := NIL" before you can use the procedures that follow. Typical operations on a list include insertion at the start of a list or at a particular position in the list. The following procedure is used for insertion before the first element in a list:

```
PROCEDURE insert (n, t: string17; VAR p: plist);
VAR
   q : plist;
BEGIN
   New (q);
   q↑.name         := n;
   q↑.telephone_no := t;
   q↑.successor    := p;
   p               := q;
END { insert };
```

With the aid of this procedure you can formulate another procedure to put a new entry at the right place in list sorted in ascending order:

```
PROCEDURE enter (n, t: string17; VAR p: plist);
BEGIN
   IF p = NIL THEN
      insert (n, t, p)
   ELSE IF n > p↑.name THEN
      insert (n, t, p↑.successor)
   ELSE IF n < p↑.name THEN
      insert (n, t, p)
   ELSE
      p↑.telephone_no := t; { new telephone number }
END { enter }
```

In the case of the "enter" procedure, it is not mandatory to use recursion. The problem could even be solved more efficiently with the aid of a repetitive statement. In both cases, however, calling the "enter" procedure triggers a linear search in the list of existing entries. The number of steps in the search for an arbitrary entry increases in proportion to the number of existing list elements.

Example 2: Binary tree

More efficient algorithms can be achieved by using tree structures. Instead of a pointer "successor", as in the list in Example 1, two pointers - ltree and rtree - are used. For each node in the tree the following situation then arises: the pointer ltree points to a subtree on the left, and rtree to a subtree on the right. All subtrees have the same data structure as the entire tree. When the tree is set up, it can be arranged that the following rules hold for each node in the tree: all names in the left subtree are smaller than the name at the current node, and all names in the right subtree are greater than that name. The advantage of this data structure resides in the generally short paths to a node with a particular name. This branching considerably simplifies the insertion of new nodes or the search for existing ones.

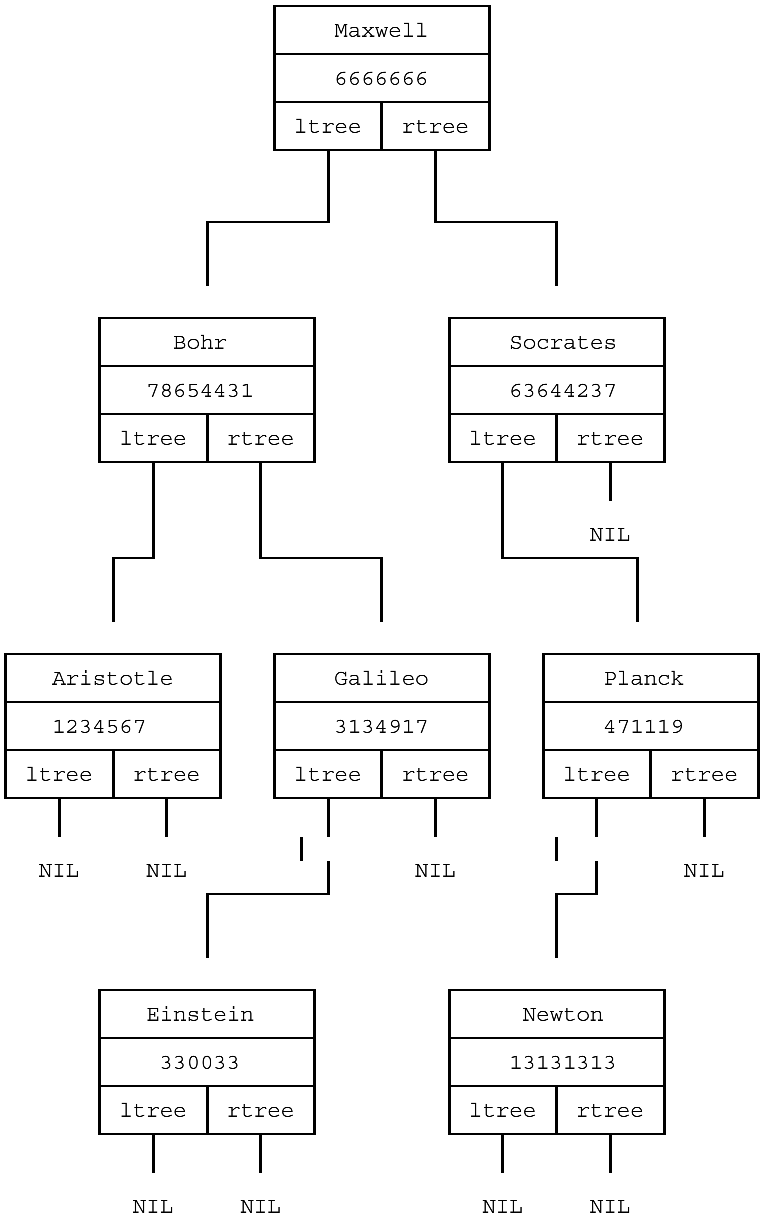


Fig. 20-1 Setting up a binary tree

One data structure typical for processing trees of this sort is defined as follows:

```
TYPE
   string17 = String[17];
   ptree    = ↑treetype;
   treetype = RECORD
                 name,
                 telephone : string17;
                 ltree,
                 rtree     : ptree;
              END;

VAR
   tree : ptree;
```

The variable "tree" should be initialized with "tree := NIL" before you use the procedu-res that follow. Typical operations on a tree structure include appending a new node to a branch ending with NIL:

```
PROCEDURE append (n, t: string17; VAR p: ptree);
BEGIN
   new (p);
   p↑.name      := n;
   p↑.telephone := t;
   p↑.ltree     := NIL;
   p↑.rtree     := NIL;
END { append };
```

Using this procedure, you can formulate another procedure to put a new entry at the right place in a binary tree:

```
PROCEDURE enter (n, t: string17; VAR p: ptree);
BEGIN
   IF p = NIL THEN
      append (n, t, p)
   ELSE IF n > p↑.name THEN
      enter (n, t, p↑.rtree)
   ELSE IF n < p↑.name THEN
      enter (n, t, p↑.ltree)
   ELSE
      p↑.telephone := t; { new telephone number }
END { enter }
```

In this case, a procedure with repetitive statements would be considerably easier to read. Since the specified data structure is recursive (it consists of a tree with similarly structured subtrees), it is more natural here to use recursive algorithms.

Another example of the use of recursion is the following procedure, which outputs a tree with the specified tree structure, sorted by name:

```
PROCEDURE printtree (p: ptree);
BEGIN
   IF p <> NIL THEN BEGIN
      printtree (p↑.ltree);
      Writeln (p↑.name, ' has the telephone number ', p↑.telephone);
      printtree (p↑.rtree);
   END
END { printtree };
```

This procedure defines a particular path through the tree. At particular points along this path an output is made - in this case whenever the left subtree has been passed through completely. The result is a printout of the names in alphabetical order with associated telephone numbers. If the three middle lines are entered in a different order, the result would be a different order of the nodes encountered along the path. For example, exchanging the lines "printtree (p↑.ltree)" and "printtree (p↑.rtree)" would produce an output in reverse alphabetical order.

Appendix

Pascal-XT Syntax

Note

The root of the syntax is "compilation unit".

```
actual-parameter
                = variable-object    | expression | procedure-name
                | function-name      | type-name
                | package-identifier | expression ":" format-denoter.

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

adding-operator = "+" | "-" | "OR" | "OR" "ELSE".

aggregate       = ARRAY-aggregate | RECORD-aggregate.

apostrophe-image
                = "''".

ARRAY-aggregate = ARRAY-type-name "(" ARRAY-aggregate-element
                              {"," ARRAY-aggregate-element} ")".

ARRAY-aggregate-element
                = expression [":" repeat-factor].

ARRAY-type      = "ARRAY" "[" index-type {"," index-type} "]" "OF"
                     component-type.

assignment      = variable-object      ":=" expression
                | function-designator  ":=" expression.

base-type       = Ordinal-type-denoter.

bit-range       = Integer-constant ".." Integer-constant.
```

```
block           = {  label-declaration-part
                   | constant-definition-part
                   | type-definition-part
                   | variable-declaration-part
                   | procedure-declaration
                   | function-declaration
                  }
                  statement-part.

buffer-variable = FILE-object "↑".

case-constant   = Ordinal-constant.

case-constant-list
                = case-constant-range {"," case-constant-range} | "ELSE".

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

case-index      = Ordinal-expression.

case-list       = case-list-element {";" case-list-element}.

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

CASE-statement  = "CASE" case-index "OF" case-list [";"] "END".

compilation-unit
                = package-specification
                | package-body
                | main-program.

component-type  = type-denoter.

compound-statement
                = "BEGIN" statement-sequence [EXCEPTION-part] "END".

conditional-statement
                = IF-statement | CASE-statement.

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

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

constant        = static-expression.

constant-definition
                = identifier "=" constant ";".

constant-definition-part
                = "CONST" constant-definition
                           {constant-definition}.
```

```
context-specification
                = WITH-list | USE-list.

control-statement
                = option [ "=" "On" | "Off" | "Restricted" |
                               character-string ].

control-variable
                = variable-identifier.

dereferenced-object
                = Pointer-object "↑".

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

digit-sequence  = digit {digit}.

directive       = "c"       | "cobol" | "external" | "fortran"
                | "forward" | "internal".

domain-type     = type-name.

ELSE-part       = "ELSE" statement.

empty-statement = .

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

EXCEPTION-part  = "EXCEPTION" statement-sequence.

EXIT-statement  = "EXIT".

exponent        = ["+" | "-"] digit-sequence.

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

factor          = primitive ["**" primitive] | "NOT" factor.

field-identifier
                = identifier ["(" offset [":" bit-range] ")"].

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

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

field-selector-identifier
                = field-identifier.

FILE-type       = "FILE" "OF" component-type.

final-value     = Ordinal-expression.

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

FOR-statement   = "FOR" control-variable ":=" initial-value
```

```
                        ( "TO" | "DOWNTO" ) final-value "DO" statement.
```

```
format-denoter  = Integer-expression [":" Integer-expression].

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

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

fractional-part = digit-sequence.

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

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

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

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

functional-parameter-specification
                = function-heading.

GOTO-statement  = "GOTO" label.

hexadecimal-digit
                = digit|"a"|"b"|"c"|"d"|"e"|"f".

hexadecimal-digit-pair
                = hexadecimal-digit hexadecimal-digit .

hexadecimal-digit-sequence
                = hexadecimal-digit {hexadecimal-digit}.

identifier      = letter {["_"] ( letter | digit )}.

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

IF-statement    = "IF" Boolean-expression "THEN" statement
                  [ELSE-part].

imported-identifier
                = constant-identifier | type-identifier
                | variable-identifier | procedure-identifier
                | function-identifier.
```

```
index-expression
                = Ordinal-expression.

index-type      = Ordinal-type-denoter.

index-type-specification
                = identifier ".." identifier ":" Ordinal-type-name.

indexed-object
                = ARRAY-object
                  "[" index-expression {"," index-expression} "]"
                | String-object "[" index-expression "]".

initial-value   = Ordinal-expression.

INLINE-function-declaration
                = "INLINE" function-heading ";" function-block ";".

INLINE-procedure-declaration
                = "INLINE" procedure-heading ";" procedure-block ";".

integer-part    = digit-sequence.

label           = digit-sequence.

label-declaration-part
                = "LABEL" label {"," label} ";".

letter          = "a"|"b"|"c"|"d"|"e"|"f"|"g"|"h"|"i"|"j"|
                  "k"|"l"|"m"|"n"|"o"|"p"|"q"|"r"|"s"|"t"|
                  "u"|"v"|"w"|"x"|"y"|"z".

main-program    = {context-specification}
                  "PROGRAM" identifier
                  ["(" program-parameter-list ")"] ";"
                  main-program-block ".".

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

multiplication-operator
                = "*" | "/" | "DIV" | "MOD" | "AND" | "AND" "THEN".

name            = [package-identifier "."] identifier.

new-type        = enumerated-type  | subrange-type | String-type
                | Pointer-type     | structured-type

object          = constant-name            | variable-name
                | aggregate                | function-designator
                | indexed-object           | selected-object
                | dereferenced-object      | buffer-variable.

offset          = Integer-constant.

option          = identifier.
```

```
package-body
                = {context-specification}
                  "PACKAGE" "BODY" package-identifier
                  ["(" program-parameter-list ")"] ";"
                  {  constant-definition-part
                   | type-definition-part
                   | variable-declaration-part
                   | procedure-declaration-part
                   | function-declaration-part
                  }
                  statement-part ".".

package-specification
                = {context-specification}
                  "PACKAGE" identifier
                  ["(" program-parameter-list ")"] ";"
                  {  constant-definition-part
                   | type-definition-part
                   | variable-declaration-part
                   | procedure-heading  ";" [directive ";"]
                   | function-heading ";" [directive ";"]
                   | "ENTRY" procedure-heading  ";"
                   | "ENTRY" function-heading ";"
                   | INLINE-procedure-declaration
                   | INLINE-function-declaration
                  }
                  "END" ".".

packed-conformant-array-schema
                = "PACKED" "ARRAY" "[" index-type-specification "]"
                  "OF" type-name.

pointer-type    = "↑" domain-type.

primitive       = unsigned-constant            | bound-identifier
                | "(" expression ")"           | set-constructor
                | qualified-set-constructor    | object.

procedural-parameter-specification
                = procedure-heading.

procedure-call  = procedure-name [actual-parameter-list].

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

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

procedure-identification
                = "PROCEDURE" procedure-identifier
                  [formal-parameter-list].

program-parameter-list
                = identifier-list.

pseudocomment   = "{$" control-statement
                    {"," control-statement} "}".

qualified-set-constructor
                = SET-type-name "(" set-constructor ")".

RECORD-aggregate
                = RECORD-type-name "(" expression {"," expression} ")".

RECORD-section
                = field-identifier-list ":" type-denoter.

RECORD-type     = "RECORD" field-list "END".

RECORD-variable-list
                = RECORD-variable-object
                  {"," RECORD-variable-object}.

relational-operator
                = "=" | "<>" | "<" | ">" | "<=" | ">=" | "IN".

repeat-factor   = Integer-constant.

REPEAT-statement
                = "REPEAT" statement-sequence
                  "UNTIL" Boolean-expression.

repetitive-statement
                = REPEAT-statement
                | WHILE-statement
                | FOR-statement.

result-type     = type-name.

RETURN-statement
                = "RETURN".

selected-object
                = RECORD-object "." field-identifier
                | field-designator-identifier.

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

SET-type        = "SET" "OF" base-type.
```

```
sign             = "+" | "-".

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

simple-statement
                 = empty-statement| assignment
                 | procedure-call | GOTO-statement
                 | EXIT-statement | RETURN-statement.

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

statement        = [label ":"]
                   (   simple-statement     | conditional-statement
                     | repetitive-statement | compound-statement
                     | WITH-statement ).

statement-part   = compound-statement.

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

string           = "'" {character-string-element } "'"
                 | "#'" {hexadecimal-digit-pair} "'".

string-element   = apostrophe-image | character.

string-length    = Integer-constant .

string-type      = String-identifier "[" string-length "]".

structured-type
                 = ["PACKED"] unpacked-structured-type.

subrange-type    = Ordinal-constant ".." Ordinal-constant.

tag-field        = field-identifier.

tag-type         = Ordinal-type-name.

term             = factor {multiplication-operator  factor}.

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

type-definition-part
                 = "TYPE" type-definition {type-definition}.

type-denoter     = type-name | new-type.

unpacked-conformant-array-schema
                 = "ARRAY" "[" index-type-specification
                           {";" index-type-specification} "]" "OF"
                      ( type-name | conformant-array-schema ).

unpacked-structured-type
                 = ARRAY-type | RECORD-type | SET-type | FILE-type.
```

```
unsigned-constant
                = constant-name
                | unsigned-integer-number
                | unsigned-real-number
                | character-string
                | "NIL".

unsigned-integer-number
                = digit-sequence | "#" hexadecimal-digit-sequence.

unsigned-real-number
                = integer-part "." fractional-part ["e" exponent]
                | integer-part "e" exponent.

USE-list        = "FROM" package-identifier "USE"
                     imported-identifier
                     {"," imported-identifier} ";".

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

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

variable-conformant-array-specification
                = "VAR" identifier-list ":" conformant-array-schema.

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

variable-declaration-part
                = "VAR" variable-declaration
                       {variable-declaration}.

variable-parameter-specification
                = "VAR" identifier-list ":" type-name.

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

variant-part    = "CASE" variant-selector "OF"
                     variant {";" variant} .

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

WHILE-statement = "WHILE" Boolean-expression "DO" statement.

WITH-list       = "WITH" package-identifier
                    {"," package-identifier} ";".

WITH-statement  = "WITH" RECORD-variable-list "DO" statement.
```

```
word-symbol     = "AND"   | "ARRAY"   | "BEGIN"   | "BODY"   | "CASE" |
                  "CONST"   | "DIV"   | "DO"   | "DOWNTO"   | "ELSE"   |
                  "END"   | "ENTRY"| "EXCEPTION" | "EXIT" | "FILE" |
                  "FOR" | "FROM"   | "FUNCTION"   | "GOTO"   | "IF"   |
                  "IN"   | "INLINE"   | "LABEL"    | "MOD"   | "NIL"   |
                  "NOT"   | "OF"   | "OR"   | "PACKAGE"   | "PACKED"   |
                  "PROCEDURE"   | "PROGRAM"   | "RECORD" | "REPEAT" |
                  "RETURN"   | "SET"   | "THEN"   | "TO"   | "TYPE"    |
                  "UNTIL"   | "USE"   | "VAR"   | "WHILE" | "WITH".
```

Required Identifiers

	Identifier	Section containing the definition
Constants:	Break_Error	5.2
	Case_Error	5.2
	Elab_Error	5.2
	Eof_Error	5.2
	False	5.2
	File_Error	5.2
	Index_Error	5.2
	Long_Maxint	5.2
	Long_Maxreal	5.2
	Long_Minint	5.2
	Long_Minreal	5.2
	Maxint	5.2
	Maxreal	5.2
	Memory_Error	5.2
	Minint	5.2
	Minreal	5.2
	Numeric_Error	5.2
	Open_Error	5.2
	Pointer_Error	5.2
	Range_Error	5.2
	Read_Error	5.2
	Short_Maxint	5.2
	Short_Maxreal	5.2
	Short_Minint	5.2
	Short_Minreal	5.2
	Set_Error	5.2
	String_Error	5.2
	System_Error	5.2
	True	5.2
	Variant_Error	5.2
Types:	Any_File	6.5.1
	Any_Type	6.5.3
	Boolean	6.2.4
	Char	6.2.3
	Integer	6.2.1
	Long_Integer	6.2.1
	Long_Real	6.2.2
	Pointer	6.5.2
	Real	6.2.2
	Short_Integer	6.2.1
	Short_Real	6.2.2
	String	6.3.2.2
	Text	6.3.5.2
Variables:	Input	11.5
	Output	11.5

Subprograms:

Identifier	Section containing the definition
Abs	15.4
Alignof	15.9
Arctan	15.4
Assignfile	15.1
Bitsizeof	15.9
Card	15.6
Chr	15.6
Concat	15.3
Convert	15.10
Cos	15.4
Delete	15.3
Dispose	15.2
Elaborate	15.12
Eof	15.1
Eoln	15.1
Error_Number	15.11
Exp	15.4
First	15.9
Get	15.1
Insert	15.3
Last	15.9
Length	15.3
Ln	15.4
Long	15.5
Long_Round	15.5
Long_Trunc	15.5
Mark	15.2
Maxlength	15.9
New	15.2
Odd	15.7
Offsetof	15.9
Ord	15.6
Pack	15.8
Page	15.1
Position	15.3
Pred	15.6
Put	15.1
Raise	15.11
Read	15.1
Readln	15.1
Readstring	15.3
Release	15.2
Reset	15.1
Rewrite	15.1
Round	15.5
Setmax	15.9
Setmin	15.9
Short_Round	15.5
Short_Trunc	15.5
Sin	15.4
Sizeof	15.9
Sqr	15.4
Sqrt	15.4

Identifier	Section containing the definition
Substring	15.3
Succ	15.6
Trunc	15.5
Unpack	15.8
Write	15.1
Writeln	15.1
Writestring	15.3

Meaning of the Word Symbols and Special Symbols

Word symbol		Section containing the definition
AND	Boolean operators	9.3.2
ARRAY	ARRAY types	6.3.1
BEGIN	statement part	12.1
	compound statement	10.2
CASE	variants	6.3.3.1
	CASE statement	10.3.2
CONST	constant definition part	5.1
DIV	arithmetic operators	9.3.1
DO	WHILE statement	10.4.2
	FOR statement	10.4.3
	WITH statement	10.5
DOWNTO	FOR statement	10.4.3
ELSE	IF statement	10.3.1
	CASE statement	10.3.2
END	statement part	12.1
	compound statement	10.2
	CASE statement	10.3.2
	RECORD types	6.3.3
FILE	FILE types	6.3.5
FOR	FOR statement	10.4.3
FUNCTION	function declarations	8.2
GOTO	GOTO statement	10.1.4
IF	IF statement	10.3.1
IN	set operators	9.3.3
LABEL	label declaration part	4
MOD	arithmetic operators	9.3.1
NIL	constant definition part	5.1
NOT	Boolean operators	9.3.2
OF	variants	6.3.3.1
	CASE statement	10.3.2
OR	Boolean operators	9.3.2
PACKED	packed types	6.3
PROCEDURE	procedure declarations	8.1
PROGRAM	program heading	11.1
RECORD	RECORD types	6.3.3
REPEAT	REPEAT statement	10.4.1
SET	SET types	6.3.4
THEN	IF statement	10.3.1

TO	FOR statement	10.4.3
TYPE	type definition part	6.1
UNTIL	REPEAT statement	10.4.1
VAR	variable declaration part	7.1
	variable parameters	8.5.2
	conformant array parameters	8.5.4
WHILE	WHILE statement	10.4.2
WITH	WITH statement	10.5

Word symbols 1. Word symbols introduced in Pascal-XT		Section containing the definition
BODY	packages and programs	11.2
ENTRY	packages and programs	11.2
EXCEPTION	exception handling	14
EXIT	EXIT statement	10.1.5
FROM	relations between packages	11.2
INLINE	procedure declarations	8.3
	functions declarations	8.3
PACKAGE	packages and programs	11.2
RETURN	RETURN statement	10.1.6
USE	relations between packages	11.3

2. Standard word symbols with additional semantic meanings in PASCAL-XT:

AND THEN	Boolean operators	9.3.2
ELSE	variants	6.3.3.1
	CASE statement	10.3.2
OR ELSE	Boolean operators	9.3.2
WITH	relations between packages	11.3

Special symbols

Symbol	Meanings
+	addition, set union
–	subtraction, set difference
*	multiplication, set intersection
**	raised to the power of
/	division
:	colon, defined variable, separator
=	is equal to
	defined type, defined constant
<	is less than
	is a genuine subset of
>	is greater than
	is a genuine subset of
<>	is not equal to
<=	is less than or equal to, is a subset of
>=	is greater than or equal to, is a superset of
[]	square brackets, open and close
()	parentheses, open and close
{}	curly brackets (braces), open and close (comment)
:=	variable on left is assigned value on right
•	decimal point, separator
,	comma, separator
;	semicolon, separator
..	2 periods, range indicator
↑	up arrow, dereferencing

Extensions to Standard in Pascal-XT

All extensions in Pascal-XT to Standard Pascal (DIN 66 256) can be reported when a program is compiled by entering the pseudocomment

```
{$ STANDARD = ON }
```

in the Pascal source program (see chapter 16).

- Word symbols and special symbols

BODY	ENTRY	EXCEPTION	EXIT	FROM
INLINE	PACKAGE	RETURN	USE	

  "**"
- Identifiers
  Identifiers may contain underscore characters.
- Directives
  C Cobol Fortran External Internal
- Numbers
  Integer values may be specified in hexadecimal form.
- Character strings
  Character strings may be specified in hexadecimal form.
  Empty character string.
- Comments
  Pseudocomments for controlling the compiler.
- Declarations
  Declarations may be entered in any order.
- Constants
  – NIL may be located on the right-hand side of a constant declaration.
  – Static expressions on the right-hand side of a declaration.
  – Additional required (predefined) constants.
- Types
  – Required type identifiers:

Short_Integer	Long_Integer	Short_Real	Long_Real
Pointer	Any_File	Any_Type	String

  – RECORD type
    Specifications for memory representation of RECORD fields.
    ELSE part in variant part of a RECORD type.
    Specifications of areas in the case constant list.
- Procedure and function declarations

- Repetition of parameter lists in identifications.
- INLINE subprograms.
- ENTRY procedures in packages.
- Result type of functions may be any type.

- Expressions
  - Shortcut operators OR ELSE and AND THEN.
  - Exponential operator "**".
  - Symmetrical set difference "/".
  - Genuine subset relations "<" and ">".
  - Set constructor with specification of SET type.
  - Static expressions may take the place of constants.
  - ARRAY aggregates and RECORD aggregates.
  - Indexing of structured values.
  - Selection of components of structured values.
  - Selecting, indexing and dereferencing of function results.
  - Comparison of character strings of different lengths.
- Statements
  - Compound statement with exception handling part.
  - EXIT statement for abandoning a loop.
  - RETURN statement for abandoning a block.
  - CASE statement.
    - ELSE as case constant.
    - Specification of areas in the case constant list.
- Package concept
  - Package specification and package bodies.
  - Context specifications WITH and USE.
  - Concept of private pointer type (with packages).
- Extensions to required subprograms

  New Dispose Put Read Write
  Pack Unpack
- New required subprograms

  Mark Release Assignfile
  Delete Insert Readstring Writestring Length
  Position Concat Substring
  Raise Error_Number
  Elaborate
  Long Short_Round Long_Round Short_Trunc Long_Trunc
  Setmin Setmax Card Sizeof Bitsizeof
  Alignof Offsetof First Last Maxlength
  Convert

List of Runtime Errors

All errors described in the manual are sorted by error class in the following tables. This makes it easy to ascertain which errors may occur in a specific error class.

Each error text is accompanied by references to this Manual in parentheses and references to the the Pascal standard in square brackets insofar as they are mentioned there.

NUMERIC_ERROR

- In an expression of the form x/y, y = 0 (9.3.1), [D.44].
- In an expression of the form i DIV j, j = 0 (9.3.1), [D.45].
- In an expression of the form i MOD j, j <= 0 (9.3.1), [D.46].
- The result of an arithmetic operation does not lie in the value range of the result type (9.3.1), [D.47]:
  - For operands of type Short_Integer or a subrange thereof, the type is Integer.
  - For operands of type Long_Integer or a subrange thereof, the type is Long_Integer.
  - For operands of type Short_Real, the type is Short_Real
  - For operands of type Long_Real, the type is Long_Real
- For Abs(x), the function result does not lie in the value range of the result type (Integer or Long_Integer or Short_Real or Long_Real) (15.4).
- For Sqr(x), the function result does not lie in the value range of the result type (Integer or Long_Integer or Short_Real or Long_Real) (15.4), [D.32].
- The result of Exp(x) does not lie in the value range of the result type (Short_Real or Long_Real) (15.4).
- For Ln(x), x <= 0 (15.4), [D.33].
- For Sqrt(x), x < 0 (15.4),D.34].
- The result of Trunc(x) or Short_Trunc(x) or Long_Trunc(x) does not lie in the value range of the result type (Integer or Short_Integer or Long_Integer) (15.5), [D.35].
- The result of Round(x) or Short_Round(x) or Long_Round(x) does not lie in the value range of the result type (Integer or Short_Integer or Long_Integer) (15.5), [D.36].

- In an expression of the form x**n, either
  - x is of an Integer type and n < 0, or
  - x = 0 or x = 0.0 and n <= 0 (9.3.1).
- In an assignment, the value of the expression (right-hand side) of type Long_Real does not lie in the value range of the variable access or the function identifier (left-hand side) of type Short_Real (10.1.2).
- With value parameter passing, the value of the actual parameter of type Long_Real does not lie in the value range of the formal parameter of type Short_Real (8.5.1).
- In an aggregate, the value of an aggregate element of type Long_Real does not lie in the value range of the associated aggregate component of type Short_Real (9.5).
- When reading from a non-textfile with Read(f,v), the value of the buffer variable f↑ of type Long_Real does not lie in the value range of the variable v of type Short_Real (15.1).
- When writing to a non-textfile with Write(f,a), the value of the expression a of type Long-Real does not lie in the value range of the buffer variable f↑ of type Short_Real (15.1).

RANGE_ERROR

- In an assignment, the value of the expression (right-hand side) of an Ordinal type does not lie in the value range of the type possessed by the variable access or function identifier (left-hand side) (10.1.2) [D.49].
- With value parameter passing, the value of the actual parameter of an Ordinal type does not lie in the value range of the type possessed by the formal parameter (8.5.1), [D.7].
- In an aggregate, the value of an aggregate element of an Ordinal type does not lie in the value range of the type of the associated aggregate component (9.5).
- When reading from a non-textfile with Read(f,v), the value of the buffer variable f↑ of an Ordinal type does not lie in the value range of the type possessed by the variable v (15.1), [D.17].
- When writing to a non-textfile with Write(f,a), the value of the expression a of an Ordinal type does not lie in the value range of the type of the buffer variable f↑ (15.1), [D.18].

– The character value Chr(x) does not lie in the value range of the Char type (15.6), [D.37].

- The result of Succ(x) does not lie in the value range possessed by the type of x (15.6), [D.38].
- The result of Pred(x) does not lie in the value range of the type of x (15.6), [D.39].

  When the statement in a FOR statement is executed, the initial or final value of the FOR statement does not lie in the value range of the type possessed by the control variable (10.4.3), [D.52, D.53].
- When reading an Integer number from a textfile (with Read(f,x)) or from a character string expression (with Readstring(s,x)), the value of the number does not lie in the value range of the variable x and no Read_Error (see below) has occurred (15.1, 15.3), [D.55].
- With Write(f,a:l1:l2) or Writestring(s,a:l1:l2), the total output length l1 < 1 or the number of digits after the decimal point l2 < 1. With Write(f,a:l1) or Writestring(s,a:l1), the total output length l1 < 1 or l1 < 0 if a is of a String type (15.1), [D.58].

SET_ERROR

- In an assignment, the value of the expression (right-hand side) of a SET type does not lie in the value range of the type possessed by the variable access or function identifier (left-hand side) (10.1.2), [D.50].
- With value parameter passing, the value of the actual parameter of a SET type does not lie in the value range of the type possessed by the formal parameter (8.5.1), [D.8].
- In an aggregate, the value of an aggregate element of a SET type does not lie in the value range of the type of the associated aggregate component (9.5).
- When reading from a non-textfile with Read(f,v), the value possessed by the buffer variable f↑ of a SET type does not lie in the value range of the type of the variable v (15.1), [D.17].
- When writing to a non-textfile with Write(f,a), the value of the expression a of a SET type does not lie in the value range of the type possessed by the buffer variable f↑ (15.1), [D.18].
- In a set constructor, the value of the member designator does not lie in the value range of the base type of the set constructor (9.4).
- With Setmin(s) or Setmax(s), the value of the expression s is equal to the empty set (9.4, 14.3.4, 15.6).

STRING_ERROR

- In an assignment, the actual length of the character string value of the expression (right-hand side) is greater than the maximum length of the String type possessed by the variable or function identifier (left-hand side) (10.1.2).
- In an assignment, the actual length of the character string expression of a String type (right-hand side) is not equal to the length of the fixed character string type possessed by the variable access or *function* identifier (left-hand side) (10.1.2).
- With value parameter passing, the actual length of the character string of the actual parameter is greater than the maximum length of the String type of the formal parameter (8.5.1).
- With value parameter passing, the actual length of the character string of the actual parameter is not equal to the length of the fixed String type possessed by the formal parameter (8.5.1).
- In an aggregate, the actual length of a character string of an aggregate element is greater than the maximum length of the String type of the associated aggregate component (9.5).
- In an aggregate, the actual length of a character string of an aggregate element (of a String type) is not equal to the length of the associated aggregate component (of a fixed character string type) (9.5).
- When reading from a non-textfile with Read(f,v), the actual length of the character string of a String type in the buffer variable f↑ is greater than the maximum length of the type possessed by the String variable v (15.1).
- When reading from a non-textfile with Read(f,v), the actual length of the character string of a String type in the buffer variable f↑ is not equal to the length of the fixed character string type possessed by the variable v (15.1).
- When writing to a non-textfile with Write(f,a), the actual length of the character string is greater than the maximum length of the String type of the buffer variable f↑ (15.1).
- When writing to a non-textfile with Write(f,a), the actual length of the character string a of a String type is not equal to the length of the buffer variable f↑ of a fixed character string type (15.1).
- With Read(f,v) or Readstring(s,v), the maximum length of the String variable v is less than the length of the character string entered

(15.1, 15.3).

- With Readstring(a,v1,...,vn), the character string expression a does not contain as many characters as requested by the read parameters v1, ...,vn (15.3).
- With Writestring(s,p1,...,pn), the maximum length of the String variable s is less than the character string formed from the write parameters p1,...,pn (15.3).
- With Delete(s,i,l), i<1 or l<0 or (i+l-1) > Length(s) (15.3).
- With Insert(s1,s2,i), i<1 or Length(s2) + Length(s1) > Maxlength(s2) (15.3).
- With Substring(s,i,l), i<1 or l<0 or (i+-1) > Length(s) (15.3).
- With Pack(a,i,z), the maximum length of the String variable z is too small to accommodate all of the characters from the unpacked array a starting starting at index i (15.8).

INDEX_ERROR

- When indexing an array variable, array constant, array aggregate or a function result of an ARRAY type, the value of the index expression does not lie in the value range of the index type of the ARRAY type (9.6.2), [D.1].
- When indexing a variable, constant or function result of a String type, the value of the index expression is less than 1 or greater than the actual length of the character string (9.6.2).
- With a conformant array parameter, the index type of the actual parameter is not a subrange of the index type of the conformant array schema (8.5.4), [D.59].
- With Pack(a,i,z), the value of the expression i does not lie in the value range of the index type of the unpacked array parameter a (15.8), [D.26].
- With Pack(a,i,z), the index range of a is exceeded while moving the components from the unpacked array a to the packed array z starting at index i (15.8), [D.28].
- With Unpack(z,a,i), the ordinal value of expression i does not lie in the value range of the index type of the unpacked array parameter a (15.8), [D.29].
- With Unpack(z,a,i), the specified area in the unpacked array a starting at index i is too small to accommodate all the components of the packed array z (15.8), [D.31].
- With Unpack(z,a,i), the character string expression z has more characters

than can be moved to the unpacked array a starting at index i (15.8).

POINTER_ERROR

- With pointer dereferencing, the value of the variable, constant or function result of a Pointer type is equal to the NIL value (9.6.4), [D.3].
- With a Dispose(p) call, p has the NIL value (15.2), [D.23].
- With a Release(p) call, the identifying value of p was not created by a Mark call (15.2).

VARIANT_ERROR

- An inactive variant of a variable, constant, aggregate or function result of a RECORD type was accessed (9.6.3), [D.2].

CASE_ERROR

- In a CASE statement, there is no CASE constant corresponding to the value of the case index, and no ELSE alternative is specified (10.3.2), [D.51].

FILE_ERROR

- Before calling Put(f), Write(f,...), Writeln(f,...) or Page(f), file f was not opened for writing (15.1), [D.9].
- Before calling Put(f), Write(f,...), Writeln(f,...) or Page(f), file f is undefined (15.1), [D.10].
- Before calling Put(f), Write(f,...), Writeln(f,...) or Page(f), the actual file position is not the end-of-file position, i.e. Eof(f) is False. This error can only occur in conjunction with [D.9]. See [D.11].
- Before calling Get(f) or Read(f,...), file f was not opened for writing (15.1), [D.14].
- Before calling Get(f) or Read(f,...), file f is undefined (15.1), [D.15].
- Before calling Read(f,...), the buffer variable f↑ of file f is undefined. This error can only occur in conjunction with [D.15]. See (15.1), [D.57].
- Before calling Eof(f), file f is undefined (15.1), [D.40].
- Before calling Eoln(f), file f is undefined (15.1), [D.41].
- With Assignfile(f,ext), the description of the external file in the "ext" operand is invalid (15.1).

EOF_ERROR

- With a Get(f) or Read(f,...) call, the end-of-file is already reached, i.e. Eof(f) is True (15.1), [D.16].
- With a Eoln(f) call, the end-of-file is already reached, i.e. Eof(f) is True (15.1), [D.42].

OPEN_ERROR

- With a Reset or Rewrite call, the required textfile Input or Output is specified (15.1).
- With a Reset(f) call, file f is undefined (15.1), [D.13].
- With Reset(f), the file linked to f and residing outside the program cannot be opened for reading (15.1).
- With Rewrite(f), the file linked to f and residing outside the program cannot be opened for writing (15.1).

READ_ERROR

- When reading an Integer number or a Real number from a textfile (with Read(f,v)) or from a string expression (with Readstring(s,v)), the following applies:
  - the input string is syntactically errored [D.54];
  - the input string forms a number which cannot be represented internally. For Integer numbers, the value lies outside the range Long_Minint .. Long_Maxint. For Real numbers, the value lies outside the range -Long_Maxreal .. Long_Maxreal.

MEMORY_ERROR

- Program execution cannot continue due to a shortage of memory space (e.g. with a subprogram call or with New) (13.3.3, 15.2).

ELAB_ERROR

- Package initialization of a program cannot continue as loops will arise during initialization due to the use of the required procedure Elaborate (13.3.3, 15.12).

Miscellaneous errors (without error numbers)

The errors given in the table below have not been assigned error numbers. If one of these errors occurs, it is not detected, and will generally lead to follow-up errors with unpredictable effects.

- In an assignment, the type of the expression (right-hand side) is of the generic pointer type and the pointer value of the expression points to an identified variable whose type differs from the domain type of the type of the variable access or *function* identifier (left-hand side) (10.1.2).
- With value parameter passing, the type of the actual parameter is of the generic pointer type and the pointer value of the expression points to an identified variable whose type differs from the domain type of the formal parameter (8.5.1).
- In an aggregate, the type of a Pointer value is of the generic pointer type and the Pointer value of the expression points to an identified variable whose type differs from the domain type of the type possessed by the corresponding aggregate component (9.5).
- The length of a String variable is modified even though there is still a reference to a component of the String variable (9.6.2).
- With Writestring(s,p1,...,pn), one of the write parameters p1,...,pn contains a reference to the String variable s (15.3).
- With Convert(x,t) the representation in memory is not a valid value of type t (15.10).
- The variant of a RECORD variable is not active for the entire duration of each reference to each one of its components (9.6.3), [D.2].
- The file pointer of a file variable f is modified (e.g. by reading or writing) even though there is still a reference to the buffer variable f↑ (9.6.5), [D.6].
- With pointer dereferencing, the value of the variable, constant or function result of a Pointer type is undefined (9.6.4), [D.4].
- With Dispose(q), an identifying value to an identified variable is removed although there is still a reference to the identified variable (15.2), [D.5].
- With a Dispose(p) call, the value of p is undefined (15.2), [D.24].
- Before calling Dispose(p), p↑ is created by New(q,c1,...,cn) or New(q,c1,...,cn,e) or New(q,e) (15.2), [D.20].
- Before calling Dispose(p,k1,...,km), the identified variable p↑ was created by New(p,c1,...,cn), with m not equal to n (15.2), [D.21].

- With Dispose(p,c1,...,cn) or Dispose(p,c1,...cn,e), the identified variable p↑ has active variants other than those specified by the CASE constants c1 to cn (15.2), [D.22].
- Before calling Dispose(p,e), p↑ was created by New(q,a) with a not equal to e. By analogy, the same applies to Dispose(p,c1,...,cn,e) and New(q,k1,...kn,a) (15.2).
- With Dispose (p,e) or Dispose (p,c1,...,cn,e) the value of e does not lie in the value range of the index type of the corresponding ARRAY type or is less than 1 or greater than the maximum length of the corresponding String type (15.2).
- In an indexed *ARRAY* or *String* object, the *ARRAY* or *String* object was generated in short form by calling New(p,e) or New (p,c1,...,cn,e) and the value of the index expression in the indexed object is greater than e (9.6.2).
- In an indexed *String* object the value of the *String* object is undefined (irrespective of whether the indexed object occurs in an expression or e.g. as a variable access on the left-hand side of an assignment).
- In an identified variable created with New(p,c1,...,cn) or New(p,c1,...,cn,e), a variant is activated other than the one specified by the CASE constants c1 to cn (9.6.3, 15.2), [D.19].
- An identified String variable created with New(p,e) or the final component of an identified String variable created with New(p,c1,...,cn,e) is assigned a character string longer than e (15.2).
- An identified variable created with New(p,c1,...,cn), New(p,e), or New(p,c1,...,cn,e) appears intact in an expression or as the left-hand side of a value assignment, or is passed as a parameter (9.6.4, 15.2), [D.25].
- With New (p,e) or New (p,c1,...,cn,e), the value of e does not lie in the value range of the index type of the corresponding ARRAY type, or it is less than 1 or greater than the maximum length of the corresponding variable string type (15.2).
- The identifying value p passed with the Release (p) call was destroyed by another Release (q) call (15.2).
- Before calling Put(f), Put(f,c1,...,cn), Put(f,e) or Put(f,c1,...,cn,e), the buf-fer variable f↑ is undefined (15.1), [D.12].

- With abbreviated output of an array with Put(f,e) or Put(f,c1,...,cn,e), the value of the index expression e does not lie in the value range of the index type of the array (15.1).
- With abbreviated output of a variable string with Put(f,e) or Put(f,c1,...cn,e), the value of the index expression e is less than 1 or greater than the actual length of the character string (15.1).
- A variable access used as an object in an expression has an undefined value at the time the expression is evaluated (9.1), [D.43].
- The result of a function will be unpredictable after function block execution if the function identifier has not been assigned a value (8.7), [D.48].
- With Unpack(z,a,i), some component of the packed array z is undefined (15.8), [D.30].
- With Pack(a,i,z), a component of the unpacked array a is accessed although the component is undefined (15.8), [D.27].
- The identifiers of the program parameters of the main program and all associated packages do not differ pair by pair (except for Input and Output) (13.3.3).
- The names of all packages belonging to a program and the name of the main program do not differ pair by pair (13.3.3).

Implementation-defined Characteristics

The implementation-defined characteristics may differ for each Pascal-XT implementation; however, they are defined for each implementation. A program can draw on implementation-defined values or characteristics, although this may lead to different results when it runs on different implementations. One simple example of this is the output of the value Maxint, which can be Short_Maxint or Long_Maxint depending on the implementation involved.

All of the implementation-defined characteristics in this manual are specified in the User's Guides [1], [2]. The cross-references in parentheses refer to sections in this Language Reference Manual, while the numbers in square brackets refer to sections in the Pascal standard, insofar as these characteristics are mentioned there.

The characteristic given in item 16 is defined identically for all Pascal-XT implementations.

1) The values of the following required Real constants are implementation-defined (5.2):

```
Short_Minreal
Long_Minreal
Short_Maxreal
Long_Maxreal
```

2) The following required constants have implementation-defined values (5.2),[6.7.2.2]:

```
Maxint    is equal to Short_Maxint or Long_Maxint
Minint    is equal to Short_Minint or Long_Minint
Maxreal   is equal to Short_Maxreal or Long_Maxreal
Minreal   is equal to Short_Minreal or Long_Minreal.
```

3) As regards the required type identifiers Integer and Real the following is implementation-defined (6.2.1):

```
Integer is equal to Short_Integer or Long_Integer
Real    is equal to Short_Real or Long_Real.
```

4) The values of the types Short_Real and Long_Real represent implementation-defined subsets of Real numbers (6.2.2), [6.4.2.2].

5) The results of arithmetic Real operators and Real functions are approximate values of the mathematical results. The precision of these approximations is implementation-defined (6.2.2), [6.7.2.2].

6) The values of type Char are obtained by enumeration of the implementation-defined characters, and the assignment of ordinal numbers of type Integer to the character values is implementation-defined (6.2.3), [6.4.2.2].

7) The maximum length of the String type without specification of a Type

parameter is implementation-defined (6.3.2.2).

8) The size of a memory unit is implementation-defined. It may be one byte or a multiple (generally a power of two) of one byte (6.3.3.2).

9) The offset and bit-range specifications in field identifiers of a RECORD type may be subject to implementation-defined restrictions (6.3.3.2).

10) The maximum number of values of the base type of a set may be subject to implementation-defined restrictions (6.3.4).

11) The greatest ordinal value of the base type of a non-qualified set constructor is implementation-defined (9.4).

12) Which of the directives C, Cobol, Fortran, External and Internal are supported is implementation-defined (8.6).

13) Subprograms with any of the directives C, Cobol, Fortran, External and Internal may be subject to implementation-defined restrictions with regard to the kind and type and number of parameters (8.6).

14) In Pascal, the logical results of the file operations are described. The physical activities and the timing of their execution are implementation-defined (19), [6.6.5.2].

15) The definition of the external file in the required procedure Assignfile and the effect of this procedure are implementation-defined (15.1).

16) The effect of the required procedures Reset and Rewrite on one of the required textfiles Input or Output is, in line with the Pascal standard, implementation-defined.
In Pascal-XT, this characteristic is defined for all implementations: an Open_Error will occur (15.1),[6.10].

17) The default output lengths for values of an Integer type, of a Real type and of the Boolean type are implementation-defined (15.1), [6.9.3.1].

18) The way the exponent is represented (either as E or as e) and the number of decimal places for the exponent when Real values are output in floating point representation is implementation-defined (15.1), [6.9.3.4.1].

19) The representation (upper case/lower case) of Boolean values in the output is implementation-defined for each letter (15.1), [6.9.3.5].

20) The effect of the required procedure Page on textfiles is implementation-defined (15.1), [6.9.5].

21) Entry subprograms may be subject to implementation-defined restrictions (11.2.1).

22) For program parameters whose variables contain a FILE type, the assignment to

objects outside of the program is implementation-defined (11.5), [6.10].

23) The presettings of compiler options are implementation-defined (16).

Implementation-dependent Characteristics

The implementation-dependent characteristics may differ in the various Pascal implementations, and apart from the characteristic mentioned in item 8, which applies to all Pascal-XT implementations, they are **not necessarily defined**.

A program based on particular implementation-dependent characteristics does not comply with the Standard, and is errored in Pascal-XT unless the characteristic is implementation-defined (see item 8).

The cross-references in parentheses refer to sections in this manual; numbers in square brackets refer to sections in the Pascal standard.

1) The sequence in which index expressions are evaluated in an indexed variable, an indexed constant, an indexed aggregate or an indexed function designator is implementation-dependent (9.6.2), [6.5.3.2].
2) The sequence in which member designators and the expressions in member designators are evaluated in a set constructor is implementation-dependent (9.4), [6.7.1].
3) Apart from the shortcut operators OR ELSE and AND THEN, the sequence in which the operands of a dyadic operator are evaluated is implementation-dependent. The operands may be evaluated in the order they are written, in reverse order, simultaneously, or possibly not at all (9.3), [6.7.2.1].
4) The sequence in which expressions are evaluated and assigned to the components of an aggregate is implementation-dependent (9.5).
5) The sequence in which actual parameters are accessed, evaluated and assigned in a procedure statement or function designator is implementation-dependent (8.7), [6.7.3, 6.8.2.3].
6) The sequence in which the variable (right-hand side) of an assignment is accessed or the expression (left-hand side) is evaluated is implementation-dependent (10.1.2), [6.8.2.2].
7) The effect of reading a textfile for which the required procedure Page was applied during its generation is implementation-dependent. however, the effect can be defined for an implementation (15.1), [6.9.5].
8) For program parameters whose variables do not have a FILE type, the assignment to objects outside the program is implementationdependent (11.5), [6.10].
   In Pascal-XT, a variable cannot be specified in the program parameter list unless it has a FILE type.

CLOCK package

The CLOCK package supplies information on the time, date, and amount of CPU time used.

```
package CLOCK;


            (****************************************)
            (*  The body of this package is part   *)
            (*  of the Pascal-XT runtime system    *)
            (****************************************)


type

  date_string = packed array [1..12] of char;   (* ISO Date : 'yy-mm-ddjjj ' *)
  time_string = packed array [1.. 8] of char;   (* ISO Time : 'hh:mm:ss'     *)
  cpu_seconds = real;                           (* cpu time in seconds       *)


function date : date_string;
   (* returns the actual date *)

function time : time_string;
   (* returns the actual time *)

function cpu_time : cpu_seconds;
   (* returns the task's used cpu time *)

end (* package CLOCK *).
```

DATE

returns the current date in ISO format (yy-mm-ddjjj), where jjj indicates the number of days accumulated in the year.

TIME

returns the current time-of-day in ISO format (hh:mm:ss).

CPU_TIME

returns the amount of CPU time (in seconds) used since the start of the process. The CPU time is counted in the implementation-defined increment of 0.02 seconds.

References

The publications marked with an * are not published by Siemens Nixdorf Informations-systeme AG or by Siemens AG.

[1] **Pascal-XT (BS2000)**
User’s Guide

Target group
Pascal-XT users in BS2000.
Contents
Operation of the programming system and of the compiler; description of the BS2000-specific attributes of the compiler; linking and executing programs; language interfaces; runtime error messages; description of predefined packages; comparison with Pascal Version 3.

[2] **Pascal-XT (SINIX)**
User’s Guide

Target group
Pascal-XT users working under the SINIX operating system.
Contents
- Using the compiler
- Description of the SINIX-specific characteristics of the compiler
- Pascal files and file linkage to SINIX files
- Language interfaces
- The debugging aid PATH
- Description of predefined packages

[3]* Däßler/Sommer
Pascal - Einführung in die Sprache,
DIN-Norm 66256
Springer Verlag, Berlin, Heidelberg
New York, Tokio 1983/85
ISBN 3-540-12835-2

[4]* K. Jensen, N. Wirth
Pascal user manual and report
Springer Verlag, Berlin, Heidelberg
New York, Tokio 1974

[5]* Jacques Tiberghien
The Pascal Handbook
Sybex, 1982
ISBN 3-887 45-005-1

Remark
This book is especially suitable as a reference work. It offers the programmer a summary of the most common versions of Pascal, and thus the ability to compare Pascal-XT, described in this manual.

Ordering manuals

The manuals listed above and the corresponding order numbers are to be found in the *List of Publications: Data Systems*, which also tells you how to order manuals. New publications are listed in the *Druckschriften-Neuerscheinungen Datentechnik (New Publications)*.

You can arrange to have both of these sent to you regularly by having your name placed on the appropriate mailing list. Your local Siemens office will help you.

Index