PROSPERO EXTENDED PASCAL

LANGUAGE SUMMARY

Contents

Introduction

This online document forms a programmer’s guide to the Extended Pascal language. It covers all the features of the Extended Pascal standard, together with the provisions for Object Oriented programming and smaller local additions, and includes information on run-time exceptions and the provisions for handling them. While the summary will be found sufficient for most everyday purposes, the more formal printed Language Reference Manual (often abbreviated to “LRM” in this document) contains the full and detailed definition.

The descriptions in this document cover all the features provided entirely by the compiler, including the built-in procedures and functions. In addition, programs can take advantage of a large collection of routines which are available in the run-time library, but whose definitions are not built in; instead they are contained in supplied “interfaces” that can be imported when required. The descriptions of these library routines, together with background information on their use, will be found in the separate Library Definitions document.

Source text

This section describes how the basic elements of program text are constructed from letters, digits, and other characters.

Syntax elements

When analysing a source text, the compiler treats it as a stream of meaningful elements known collectively as tokens, interspersed with separators.

There are three kinds of separator: space or tab, end-of-line, and comment. Tokens are words, numbers, character strings, characters such as comma or semicolon, or character pairs. You can put a separator (for instance a comment) between any two tokens; you must put a separator (for instance a space or end-of-line) between any adjacent pair of words or numbers. It follows that any token, such as a word or number, must be on one line, and cannot contain any embedded spaces.

As an illustration, the two following programs appear identical to the compiler. Notice that the comments placed between braces in the first example are in effect ignored, and do not affect the actual meaning of the program.

```
PROGRAM Hello (output);
  { The well-known example }
BEGIN
  writeln('Hello world!');
END {of program}.

PROGRAM Hello(output);BEGIN
writeln('Hello world!');END.
```

See also:

Words   Identifiers   Word symbols

Numbers   Character strings   Comments

Special symbols

Words

A word is a group of one or more alphanumeric characters starting with a letter; underscore is treated as an alphanumeric character for this purpose. Words are used as identifiers to name program elements such as variables and procedures; some names are part of the Pascal language and are built into the compiler, some may be imported from libraries, but most are names which you give to items you have introduced yourself. There is a set of "reserved" words with fixed meanings, such as `PROGRAM`, `BEGIN`, `WHILE`, `REPEAT` and `END`. The formal term for them is word symbols, and they cannot be used as identifiers. You are allowed to re-use the built-in identifiers, though it is not usually a good idea to do so.

A word starts with a letter, after which there can be any number of further letters, digits, or underscore characters. (In some special situations, a name may also start with an underscore.) Upper and lower case letters can be used, and the compiler does not make any distinction, but mixed case is a good way to make a name more meaningful. Here are some examples of names:

```
    ThisItem  surname  PAC12  Offset  spin_doctor
```

Afterwards you can refer to the first of these names as ThisItem, which is generally the clearest, but you can use letters in different case such as THISITEM or even thiSitem if you wish.

While there is no set rule about upper or lower case, it can be helpful to keep to a convention such as all upper case for word symbols and lower or mixed case for names.

The Pascal standards only allow letters from the Latin alphabet to be included in names. This implementation will also accept for example accented letters, provided that the machine on which the software is being used recognises them, but such source programs will not as a rule be portable to other implementations.

Identifiers

In some cases, the names (or more correctly identifiers) you use in your programs are part of the Pascal language and are built into the compiler, or have already been defined in some other way, but most will be names which you give to items you introduce yourself. For example, you may need a variable to hold the thickness of some metal sheet, so you give this variable the name Thickness. Again, if a record includes a field to hold an internet address, you might give the field the name internet_addr. In your code you then use the name to identify the item.

You have a lot of freedom in choosing names, and it is worth devoting some attention to making them work for you. Remember that references to variables and record fields tend to occur more often than to types, so keep the shorter names for them without making them too cryptic.

For a complete list of reserved words see Word symbols below.

Word-symbols

A full list of word-symbols follows. Some of them are not defined in the ANSI/ISO standards, and are not treated as word-symbols if you switch the compiler to a lower language level.

ABSTRACT	AND	AND_THEN	ARRAY
BEGIN	BINDABLE	CASE	CLASS
CONST	CONSTRUCTOR	DESTRUCTOR	DIV
DO	DOWNTO	ELSE	END
EXCEPT	EXPORT	FILE	FOR
FUNCTION	GOTO	IF	IMPORT
IN	IS	LABEL	MOD
MODULE	NIL	NOT	OF
ON	ONLY	OR	OR_ELSE
OTHERWISE	PACKED	POW	PROCEDURE
PROGRAM	PROPERTY	PROTECTED	QUALIFIED
RECORD	REM	REPEAT	RESTRICTED
SET	SHL	SHR	THEN
TO	TRY	TYPE	UNTIL
VALUE	VAR	VIEW	WHILE
WITH	XOR		

Numbers

A number is a group of characters that starts with a digit and follows one of the formats described below. (Because you can give names to constant values, and the names then have the attributes of constants, numeric values written in the program text are called "numbers" rather than "constants".) Numbers can introduce values of two main types into a program: **integer** or whole-number values, and **real** or floating-point values. Integer numbers are normally written in decimal, and may be signed, for example 99, 2000, -20, 0, 107215. They must not exceed the maximum integer value (called maxint) which in this implementation is 2147483647.

There is also a form known as an extended number that allows you to introduce an integer value in a representation other than decimal. An extended number has two parts, separated by a # character. The first part defines the base as a decimal value; often this will be 16 for hexadecimal, but it can be 2 (binary) or 8 (octal), or indeed any value up to 36. The second part is the value, which may include letters when the base is greater than 10, as will be familiar from hexadecimal notation (A=10 and so on). It follows that 255, 16#FF, and 2#11111111 are alternative representations of the same value.

Real (floating-point) numbers are always represented in decimal. There may be a decimal point; if there is, it must be preceded and followed by at least one digit. An exponent can also be specified, and is introduced by a letter E. These are all floating-point numbers:

```
    1.25  0.125E1  125E-2  -5.5  2.14159
```

There is no special format for complex numbers, but you can represent them by means of a constant expression such as `cmplx(1.5,2.5)`.

Character strings

A character string in a source text is a sequence of zero, one, or more characters enclosed between apostrophes. A string of length one is of type char, other lengths are of the general string type. An apostrophe within a character string is represented by a pair of apostrophes.

Examples:

```
        'x'
        'Boyle''s Law'
        ''
```

A character string is not restricted to the set of characters used in other tokens, and can include for example accented letters, but if you are producing a program to run in console mode the difference between the ANSI and OEM character sets must be borne in mind. In the Windows environment, new console mode programs are fairly rare, but if you have inherited older programs they may have been intended to run in this mode. You can use the Workbench to edit such sources, by introducing a “pragma” into the text. There is more information on character sets and pragmas in the full manuals.

A source program is expressed in an 8-bit character set, and there is no special representation for Unicode strings. A character string will be stored in the compiled program in Unicode form if the usage requires it, in particular if it is assigned to or combined with a widestring variable or formal parameter.

Comments

You can add comments to your program almost anywhere to explain what each part is doing. Against a variable called Thickness, for example, you can add a comment to show that it is held in millimetres. A procedure or function should generally have a comment describing the conditions under which it is entered and the results obtained. Remember, though, that Pascal is a descriptive language provided that you have paid some attention to choosing meaningful names, and a good name saves adding a comment at each reference.

A comment begins with a left brace character “{” and is terminated by a right brace “}”, for example `{millimetres}`. An alternative which uses (* instead of left brace and *) instead of right brace was originally introduced for computers without the braces in their character set, but is still accepted. The comment itself can contain any text apart from right brace or the *) combination.

Special symbols

Special symbols composed of one or two non-alphanumeric characters represent operators and punctuation.

+	addition, string concatenation, set union
-	negation, subtraction, set difference
*	multiplication, set intersection
/	real division
**	real exponentiation
:=	assignment
= <>	equal, unequal
< <=	less than, less-or-equal
> >=	greater than, greater-or-equal
( )	list containers, eg parameters
[ ]	index containers, eg arrays
^	pointer dereference
::	type viewing
.	field reference etc
,	list separator
;	terminator, separator
:	list terminator etc
..	range, substring
><	set symmetric difference
=>	export/import renaming

Program structure

The structure of Pascal programs is described in the following sections. If you are new to the language, you may wish to skip initially those marked with an asterisk.

Blocks

Blocks are important ingredients of program structure. A block has two main parts:
  Definitions and declarations
  Statements (actions)
A main program consists of a heading and a block. When you define a procedure or a function, it consists of a heading and a block.

One thing to notice is that among the definitions and declarations there can be definitions of procedures and functions. That is to say, you may find a nested structure of blocks, with some procedures at the same level and others within them (big fleas and little fleas, as the saying goes). Here is an illustration of one possible arrangement:

```
 _________________________________________________
|              PROGRAM xyz (   )                  |
|    _________________________________________    |
|   |       PROCEDURE outer1 (   )            |   |
|   |_________________________________________|   |
|    _________________________________________    |
|   |       PROCEDURE outer2 (   )            |   |
|   |    _________________________________    |   |
|   |   |   PROCEDURE inner2 (   )        |   |   |
|   |   |_________________________________|   |   |
|   |                                         |   |
|   |_________________________________________|   |
|                                                 |
|_________________________________________________|
```

The program xyz directly contains the two procedures outer1 and outer2, and in turn, outer2 contains inner2. This is of course just a skeleton, with many aspects omitted. We will expand on it when discussing scope.

As well as procedures and functions, the definitions and declarations can introduce new data types and variables, constants, and labels. In a definition or declaration you give a name (an identifier) to each new item; you can only refer to a name after it has been introduced.

A name has one meaning throughout a block. In general you can only use the name after it has been introduced (but see Pointer types).

Definition parts

Within a block, you can introduce and give names to constants, data types, variables, labels, procedures and functions. The definitions are grouped into definition parts, which in classic Pascal were required to come in a specific order. In standard Extended Pascal you can put them in any order, and any part can be repeated. However, for most purposes when you define one entity in terms of another, the reference must be to a name already defined; there is a specific exception to this when defining pointer types.

A constant definition part is a group of one or more constant definitions introduced by the word symbol `CONST`. Each definition takes the form of an identifier (which will be the name of the constant), the symbol `=`, and a constant expression giving the value. The simplest constant expression just consists of a constant, but more complicated ones can be useful, for instance in building up string constants. Following is a sample of a constant definition part.

```
    CONST   limit = 10;
         tab   = chr(9);
         title = 'Name' + tab + 'Address';
```

A type definition part is a group of one or more named type definitions introduced by the word symbol `TYPE`, see programmer-defined types. A variable declaration part is introduced by the word symbol `VAR`, and names one or more variables that will be created when the block containing the declarations is activated, see variable declarations.

For a description of procedures and functions see Procedures and Functions. A group of procedure and function declarations, with no intervening definitions or declarations of other kinds, forms a procedure-and-function-declaration-part. In Extended Pascal, this grouping is only significant when considering the `forward` directive described in procedures.

You can use a label in Pascal as the destination of a GOTO statement. A label declaration part lists one or more labels introduced by the word-symbol `LABEL`; if there is more than one, they are separated by commas, and the list is terminated by semicolon. In standard Pascal labels must be numeric, but this implementation also allows conventional identifiers, for example:

```
    LABEL   99, next_item;
```

The position of each declared label must be defined by placing it in front of a statement, followed by a colon, thus:

```
    next_item:  BEGIN ...
```

Statement part

In the statement part of a block, you specify the actions to be performed when the block is activated. A main program consists of a program heading and a block, and the statement part of this block specifies the actions of the program. A procedure or function declaration consists of a heading and a block, the statement part of the block specifying the actions of the procedure or function. A statement part takes the form of a compound statement, that is, a list of statements introduced by the word-symbol `BEGIN` and terminated by the word-symbol `END`.

Scope

When there is a possibility of nesting blocks, the question of visibility arises. Which statements can access which variables, or invoke which procedures? One of the objectives is to make nested blocks self-contained, and secure from outside interference, but there is also advantage in the code of inner blocks being able to access data of the enclosing levels. We can see what happens if we expand the earlier illustration:

```
 ____________________________________________________
|              PROGRAM xyz (...)                     |
|                                                    |
|        program-level variables g,h                 |
|    ____________________________________________    |
|   |        PROCEDURE outer1 (a,b..)            |   |
|   |                                            |   |
|   |       statement part of outer1             |   |
|   |____________________________________________|   |
|    ____________________________________________    |
|   |        PROCEDURE outer2 (c,d..)            |   |
|   |                                            |   |
|   |       variables x,y of outer2              |   |
|   |    ___________________________________     |   |
|   |   |    PROCEDURE inner2 (g..)         |    |   |
|   |   |                                   |    |   |
|   |   |  statement part of inner2         |    |   |
|   |   |___________________________________|    |   |
|   |                                            |   |
|   |       statement part of outer2             |   |
|   |____________________________________________|   |
|                                                    |
|           statement part of program                |
|____________________________________________________|
```

Here, statements of the main program can access the program-level variables g and h; they can also invoke procedures outer1 and outer2. They cannot refer to anything (such as x, y or inner2) located inside outer1 or outer2.
In the statement part of outer1, the program-level variables g and h are accessible, and also the parameters a and b.
In the statement part of outer2, the variables g and h are still accessible, and so are its parameters c and d and variables x and y. It is also possible to invoke inner2.
In the statement part of inner2, variable h is still accessible, and so are c, d, x and y. However, inner2 has a parameter called g, and this name takes precedence over the program-level variable; the program-level g is not accessible.

Programs and Modules

Programs and modules are components that can be processed separately. Every Extended Pascal program must have a main program component; it can also include one or more modules. A module can be included in more than one program if it provides relevant functionality. If this is your first introduction to Pascal, you may prefer to skip the description of modules until you have some familiarity with other aspects.

Whether a program is made up simply of a main program, or contains modules, each component is processed by the compiler, and the compiled forms are presented to the linker to make an executable program. These operations are invoked from the workbench Build menu, which also allows for various options to be specified. Compiler options allow you to specify for instance that extra checks are to be introduced; linker options allow a choice between forms of executable program. For the purpose of this summary, we assume that programs will be produced in the default form, which displays results written to the standard output file in a window maintained by the Pascal run-time system. Standard output has been available in all versions of Pascal, and is a convenient means of communicating to the user of the program. There associated standard input and standard error files allowing for

Main program

The general outline of a Pascal program is:

```
    PROGRAM name (parameters);
      Definitions and declarations
    BEGIN
      Statement part
    END.
```

Here, "name" is simply the name you have chosen for the program. The definitions and declarations introduce data types and variables, also procedures and functions, which are subsidiary self-contained program actions that you can separate from the main processing. The actions of the program are specified in the "statement part", between the symbols `BEGIN` and `END`. Actions in Pascal are specified by statements, and the statement part is made up of one or more (sometimes many) statements.

If you wish to use the standard output, input or error files for notifying results or user interaction, they must be listed in the program parameters using the names **output**, **input** or **errout**. You can then address them in writeln or readln statements, for example `writeln('Mission accomplished')`. The standard output and input files do not have to specified by name, as they are defaults when no file is mentioned, but standard error must be, as in `writeln(stderr,'Mission aborted')`. See writeln, readln.

An important part of programming lies in dividing tasks into pieces of a reasonable size. In part, this allows you to keep control, but it is specially helpful when using a screen editor, in which fairly small sections of the text are visible at one time. The outline of a program might be expanded as follows with this in mind.

```
    PROGRAM name (parameters);
      <Definitions and declarations>
      PROCEDURE Initialize;
     ...
      PROCEDURE Process;
     ...
      PROCEDURE CleanUp;
     ...
    BEGIN
      Initialize;
      WHILE NOT end_of_job DO Process;
      CleanUp;
    END.
```

For more on this topic see Procedures and Functions.

Interfaces and supply

When a program includes a module, it is necessary for other components to be able to make use of the services that it provides. (This is generally the rule, at least; very occasionally there may be examples of modules that are completely self-contained.) When writing a module, the programmer gives names to items such as variables and procedures, just as in a self-contained program. Some of the names are then published, by including them in the definition of an interface, and exporting the interface. An interface also has a name, and when another component needs to use a service provided by the module, it imports the interface by name. The constituents of the interface (the variables, procedures or other items) can then be used.

When a module X exports an interface that is then imported by another module Y or by the main program, X is said to supply the other component. And just as identifiers within a component must normally be introduced before they can be used, so there must be a notional order of modules within a program in which the definition of an interface precedes any import of that interface. In practice, this is not really restrictive; in a typical program there might be one or two low-level modules supplying middle-level modules, one or more of which supply the main program. The rule places the low-level modules at the head of the supply chain and the main program at the end, and avoids the possibility of circular supply (P supplies Q supplies R supplies P, for example), which would make the independent processing of modules impossible.

For examples of export and import parts see Module heading.

Module formats

A module is made up of a module heading and a module block. These two parts may be combined or separate; as a general rule, when a module supplies procedures or functions it is more convenient to keep them separate, and when it supplies only constants, types or variables it is better to combine them. These are by no means hard and fast rules, but the reason for the preference will be seen shortly.

The overall format when heading and block are combined is like this:

```
  MODULE  modulename [(parameters)];
    <module heading>
  END;
    <module block>
  END.
```

In this form, the whole module is held in a file which is normally called **modulename.pas**. The parts shown as `<module heading>` and `<module block>` are expanded in module heading and module block.

When the heading and block are separate, the heading takes the form:

```
  MODULE  modulename[(parameters)] interface;
    <module heading>
  END.
```

and the block takes the form:

```
  MODULE  modulename implementation;
    <module block>  END.
```

With separate parts, the heading is held in a file conventionally called **modulename.hdr** and the block in a file called **modulename.pas**. The two parts are associated by having the same module name.

The purpose of the module heading is to hold all the definitions to be supplied to other components. As will be seen, when these includes procedures and functions, the full declarations are placed in the module block; because they will as a rule require more amendment and recompilation during development than the exported definitions, keeping them separate is an advantage. On the other hand, when there are none, the block is empty, and the combined form is simpler.

A less common reason for separating heading and block is that you can then have more than one implementation of the same specification. You might wish to provide normal and diagnostic versions, say, or one version that ran faster than the other but demanded more memory.

Module heading

For the overall layout of a module, see module formats. Following the `MODULE modulename ...` line, the module heading must contain an export part, introduced by the wordsymbol `EXPORT`. The export part names one or more interfaces, and the constituents they are to include; these may be constants, types, variables, procedures and functions which are to be made available for use by other components of the program. Just the names appear in the export list; they must all be given meanings later in the module heading. A simple export part might look like this:

```
EXPORT  iface = (type1, type2, proc, func);
```

A module can export more than one interface. You might wish for instance to supply some items to several other modules, and some only to the main program. The word `EXPORT` only appears once, followed by one or more interface names and lists of constituents. There is a requirement that all the interfaces in the program have distinct names.

The export list, or lists, quoted the names of constituent items; you now give the meanings of these names. One way to do this is by importing from another module, and handing on the definition. If there is an Import part it must come immediately after the export part. In any case, you can provide definitions within the module, using definition parts, and the headings (signatures) of any procedures or functions. The full declarations of these procedures and functions will be located in the module block. Note that in this context there can be no label declarations.

There will occasionally be a need for a constituent to be known by a name in other parts of the program that is different from its name within the originating module. In such cases it can be renamed when it appears in the export list by placing `=>newname` after it, for instance to export Fred renamed as Jim the list entry is `Fred=>Jim`.

As a shorthand, you can put the word `MODULE` in an export list, standing for "all the items defined in this module heading, but not imported items". This notation avoids having to update the list whenever a new item is defined.

Module block

For the overall layout of a module, see module formats. The module block may be combined with the module heading, or may be separate, in which case it starts `MODULE modulename implementation;` and from then on the format is the same.

All identifiers that are imported into, or defined in, the module heading are available for use in the module block. This is true whether or not the identifier was exported, and whether the module heading and module block are combined or separate.

The module block must contain the full declarations of any procedures or functions whose headings (signatures) were included in the module heading. In this implementation, you can repeat the signature in the full declaration, which makes the program easier to read, particularly when the module block is separate.

Aside from this requirement, you can import into the module block any interface that was not imported by the heading, you can declare additional procedures or functions, and you can introduce constants, types and variables at the outer level for use by any of the routines. Imports are specified in an import part which must precede any other declarations or definitions.

A module block can optionally contain an initialization part and/or a finalization part. An initialization part takes the form

```
  TO BEGIN DO <statements>;
```

and is obeyed at program startup before the main program is entered. A finalization part takes the form

```
  TO END DO <statements>;
```

and is obeyed after completion of the main program. More precisely, the initialization is obeyed before that of any module supplied by this module, and finalization after that of any module supplied.

Import part

You can import one or more interfaces into a block, a module heading, or a module block. Because a block makes up most of a main program, procedure or function, this means that you can import at the program level (which is what usually happens), or (more rarely) into an individual procedure or function. Import is specified in an import part; in a module heading, if there is an import part it must immediately follow the export part; and in any case an import part precedes other definitions and declarations.

An import part starts with the word `IMPORT`, followed by one or more interface names, thus:

```
  IMPORT  interface1; interface2;
```

The effect of this is to bring in all the identifiers contained in the interfaces, together with their definitions (constants, types and so on, see Interfaces and Supply). When passing information from one program component to another, this will often be just what you want to do, but for some other situations it may well not be satisfactory.

Suppose that you wish to make use of two modules that were not originally intended for use together. You need only a few of the constituents of each of their interfaces (and maybe some of the names you do not need can be confused or even clash with other names). You can use selective import to restrict the names to just those you need, for instance:

```
  IMPORT  interface1 ONLY (abc, def, gh);
       interface2 ONLY (wxyz);
```

Situations such as this occur quite regularly when a single large interface defines the contents of a library; unselective import could bring in many names that are not needed and cause confusion. Importing just the names you plan to use will avoid the problem.

Another option that may be useful when importing two unrelated interfaces is to rename one or more of the constituents. The notation is just like the export renaming described under module heading, for example:

```
  IMPORT  fuels (petrol=>gasoline);
```

Selective import and renaming can be combined:

```
  IMPORT  iface ONLY (abc=>p, def=>qrs, gh);
```

Because extensive renaming can sometimes result in a program text that is correct but hard for human readers to understand, another option may be considered. With qualified import you can keep the names brought in from an interface in a private scope. The import is written like this:

```
  IMPORT  iface1 QUALIFIED; iface2 QUALIFIED;
```

After this, you must precede every reference to an imported name with the interface name and a period, for instance:

```
    iface2.proc (total, iface1.nominal);
```

The result is clear and explicit but more verbose, and the advantage will vary from one situation to another.

Libraries

An important use of Extended Pascal modules is in the construction of libraries. The compiled object files from the modules which are to make up a library can be collected by a librarian program, such as ProLib. In addition, the user of the library will need collected interface definitions or "prototypes". From these definitions, the user can obtain constants and types, as well as the signatures of procedures and functions.

You can use a module to collect and amalgamate interface information. This module might contain a few definitions such as an identification of the library version, but most of the material will simply be imported and re-exported, possibly with some selection or renaming.

In this implementation, there is an additional shorthand to assist in library construction. When an interface is imported `QUALIFIED`, the interface name can be quoted in an export list, with the meaning "all the details imported from this interface, without any renaming or selection".

Procedures and Functions

These are fundamental building-blocks of Pascal programs, hinted at in the description of a Main program. A range of built-in procedures and functions are provided as part of the language, but we are concerned here with defining your own to structure your programs.

To start with, let us look at the circumstances in which they help you. For one thing, they allow you avoid repetition. When you have a subsidiary task to be performed in more than one part of your program, you can formulate it once and save code. But another excellent reason is that you can organise and structure any non-trivial program in a way that allows you (indeed, encourages you) to concentrate on one part at a time. Divide and rule is how many people express it.

We will look further at writing procedures and functions that form parts of your own programs, but another aspect is just as significant. They allow you to use re-use your own code, and to employ code produced by others. Some of the built-in features of the language are accessed "as if" they are procedures or functions, and the libraries supplied with the compiler contain code to perform other useful tasks. You can build libraries of your own handy algorithms, and when you have discovered the essentials of object-oriented programming at least some of your libraries may well take the form of classes.

Procedures and functions have important features in common, but they are used in rather different ways. A procedure is a section of code that performs a sub-task; it is activated by a procedure statement. Control is passed to the procedure, and on completion normally returns. By means of parameters it can be given different data to work on at each activation. A function evaluates a result; it is activated from an expression, and returns the result as an ingredient of the expression. Again, it will in most cases have one or more parameters that provide different data as input to the process each time.

Apart from those which form part of the Pascal language, the name and "signature" of any procedure or function must be known before it can be used. The definition may be part of the program or module, or the details may be imported as a constituent of an interface.

Procedures

The definition of a procedure takes the form of a heading and a block. The heading starts with the word `PROCEDURE` and the name by which it is to be known; if the name tells what the procedure does, it makes the program much easier to read. There may then be a list of parameters to modify the action of the procedure or allow it to return results. When there is a parameter list, it is enclosed in parentheses. Following are some examples of procedure headings:

```
    PROCEDURE Initialise
    PROCEDURE MakeNewCustomerRec
            (name: string; ref: integer;
             VAR NewRec: PCustomerRec)
    PROCEDURE OpenDatabase (VolId: integer)
```

In a procedure declaration, the commonest use, the heading is followed by a semicolon and a block. The block contains definitions of any constants, variables or other items (including other procedures or functions) which are needed by the procedure but not outside it. Then there is a "statement part" that specifies the actions of the procedure. The statements in the statement part can refer to the parameters, and to any items introduced by definitions or declarations in the block; they can also refer to items such as constants or variables introduced in the containing block or blocks (unless the names of such items have been hidden - see Scope).

A procedure is activated by a procedure statement, which consists of the procedure name and, if the procedure has a parameter list, corresponding actual parameters. See Parameters.

Procedure headings occur in other situations than in the procedure declarations described above. Suppose for instance that you wish to write two procedures A and B which may call one another. The need to avoid forward references would forbid either from being the first, but you can introduce one with a forward declaration consisting of its heading and the directive forward, thus:

```
    PROCEDURE B ( ... ) forward;
    PROCEDURE A;
      ...
    PROCEDURE B ( ... );
      ...
```

The first occurrence of the heading of B enables the compiler to process correctly a call of B within the body of A. There must be no CONST, TYPE, VAR or LABEL definitions between the forward heading of B and the full declaration. The Pascal standards do not allow repetition of the parameter list when the heading of B comes again, but this implementation allows it - see Signatures.

Other situations in which the heading of a procedure appears without the block are when the procedure is exported from a module (see Module heading) and in the definition of a class as described in the introduction to Object Oriented programming.

Functions

Many of the remarks about procedures apply also to functions. A function is composed of a heading and a block, it may have parameters, and the heading or signature can appear separately as a forward declaration, or in a module heading or class type definition. There are however differences. In the first place, a function produces a result, and the heading includes the type of this result. Also, while a procedure is activated by means of a procedure statement, a function is activated by being named in an expression, and the result it returns becomes an operand in the expression.

The following examples show the form of function headings:

```
    FUNCTION area (a,b,c: real): real;
    FUNCTION Daylight (ts: TimeStamp): integer;
    FUNCTION NewCustomerRec (name: string; ref: integer)
                    = NewRec: PCustomerRec;
```

In the body of the first of these functions (which we can assume is to compute the area of a triangle having sides length a, b and c) a value must be given as though to a variable called area; this is the value that the function will return. More than one value may be assigned during the execution of a function, and the last will be the value returned. In the second function, similarly, a value must be assigned to Daylight (being say the minutes between sunrise and sunset on the date given by the TimeStamp parameter). In the third example, the result variable is given the name NewRec, different from the name of the function itself; this is a feature of Extended Pascal that allows the result to be referenced within the function. (Using the function name implies the function calling itself.)

While many functions produce results of simple types such as integer or char, it is also possible to define a function that returns a record or an array, provided it does not contain any embedded file.

Parameters

Parameters allow you to provide different data to a procedure or function at each call. In the heading of the routine, there is a list giving the names and types of formal parameters. The names are local to the routine, that is, they can be referenced by statements within the routine but not from outside; the types must be named types, not new type definitions. At each call, a matching list of actual parameters must be supplied. For example, the procedure:

```
  PROCEDURE Famous (name: string; born,died: integer)
```

might be activated by these calls:

```
    Famous ('Blaise Pascal', 1623, 1662);
    Famous ('Isaac Newton', 1642, 1727);
    Famous ('Alfred, Lord Tennyson', 1809, 1892);
```

Parameters that pass data to a procedure can be value parameters or variable parameters; because the latter are introduced by the word symbol VAR they are often referred to as VAR parameters. The majority of parameters are of these two kinds, but Pascal also provides two other forms, procedural parameters and conformant array parameters.

Value parameters

With a value parameter, you pass the value of an expression to the procedure or function being called. The actual parameter can be a constant or variable, or the result returned by a function, or when appropriate to the type it can be the outcome of a computation. The requirement is that the type of the expression must be assignment compatible with the type of the formal parameter. The parameter is in effect a local variable which is given the value of the expression as part of the process of implementing the call.

For example, the fictitious function:

```
  FUNCTION match (a,b: real): Boolean
```

has two value formal parameters a and b, and returns true if their absolute values agree within one percent of the average. The function might be called in several ways:

```
    IF match(x, 0.5) THEN ...
    WHILE match(sqr(sin(v+e))+sqr(cos(v-e)), 1) DO ...
    assert(match(p1,p2), 'Mismatch');
```

In these calls, you can see some of the possible kinds of actual parameter, including the integer constant 1 which is assignment compatible with real type. (See assert.)

A formal value parameter of string type can be either one defined as a named type with a specified capacity, or the schema name string, for example:

```
  PROCEDURE PrintName(name: namestring; address: string);
```

An actual parameter corresponding to name must not exceed the length defined as namestring, whereas address can accept effectively unlimited length (to be precise, not exceeding 32760). Again, the actual parameters are string expressions:

```
    PrintName(FamilyName+', '+FirstName,
           StreetAddr+'/'+Town+'/'+PostCode);
```

You can pass structures such as records and arrays as value parameters. The corresponding actual parameter can then be an existing variable or a constructor. If the actual is an existing record or array, the implication is that a copy of the structure is made as part of the process of calling and entering the procedure or function. To allow the routine to examine a structure without "threatening" it you can use a `PROTECTED VAR` parameter (see VAR parameters).

A formal value parameter can be designated `PROTECTED`. There is less obvious advantage than with VAR parameters, but it can be useful as indicating that the routine will not modify the value passed to it.

The rule of assignment compatibility does not allow a file (or indeed any structure which contains a file) to be passed as a value parameter. A file must always be passed as a VAR parameter.

VAR parameters

While a value parameter allows the value of an expression to be passed to a procedure or function, a variable (VAR) parameter must be matched by an actual which is a variable acess. This phrase describes a reference to a single variable, or array element, or record field, or pointer domain, or combination of these. The access, however arrived at, must be the same type as the formal parameter. For example, in the heading:

```
  PROCEDURE MakeNewCustomerRec (name: string; ref: integer;
                          VAR NewRec: PCustomerRec)
```

copied from Procedures, the NewRec is a VAR parameter through which the procedure returns a pointer. In a call of this procedure, the actual parameter corresponding to NewRec must be a variable (or element or field) of the type PCustomerRec.

That example assumes a naming convention in which PCustomerRec is used for the type of a pointer which points to a CustomerRec record. It is fairly common also to use a VAR parameter with a structured type, such as CustomerRec. The distinction is that when returning a pointer, we assume that the procedure makes a new record in the heap and gives back to the caller a pointer to the new record. However, when the formal parameter type is the record type, the caller must already have allocated a record, and statements in the procedure can refer to and modify that record.

You can define a VAR parameter that gives the procedure or function effectively read-only access to the actual parameter. The word symbol `PROTECTED` has this effect, for example:

```
  PROCEDURE PrintCustomerRec (PROTECTED VAR crec:
                                           CustomerRec)
```

The requirement that the formal and actual parameters have the "same" type means as a general rule that they must have exactly the same named type. Further, the actual parameter must not be a component of a `PACKED` structure if different storage has been allocated than for a single variable. However, in the case of schematic types (which includes strings), a somewhat different rule applies. Certainly, the same named type is still valid, but for example two types separately defined as "string(10)" are also the "same" type - see Schema types.

Procedural parameters

With a procedural (or functional) parameter, you can pass a procedure (or function) to a called routine, rather than data to work on. The formal parameter is a procedure or function heading, when the routine is called you pass a matching actual procedure or function, and the code in the routine to which the parameter is passed can invoke the parameter.

This may sound rather involved; take an example. Suppose a procedure `SmartPrint` accepts text and formats it. With a VAR file parameter it could be passed alternative input files each time it is called, but that requires the input to be always in the form of a file. If instead the procedure has a parameter that is a function, the input could come from other sources. You could define `SmartPrint` like this:

```
  PROCEDURE SmartPrint (FUNCTION NextCh: char);
```

When SmartPrint is called, it is passed a function that returns characters. Sometimes the function in fact reads the characters from a file, but in other calls the function passed might be one that takes its data from memory (or even makes it up at random). The code in `SmartPrint` simply calls `NextCh` to produce characters.

The rules for matching formal and actual procedural parameters are set out in LRM 8.2.4.

Conformant array parameters

Conventional array types in classic Pascal are fixed in size, and a procedure that processes arrays is of restricted use if it can only accept as parameter the same size array at each call. To avoid this difficulty, the original Pascal standard had an optional feature called "conformant array parameters". The Extended Pascal standard introduced schema types, which among other things allow parameters of various sizes to be passed. In the conformant array option, the index bounds of the actual parameter are automatically supplied, and can be used by the statements of the called routine. The details are set out in LRM 8.2.5.

Signature

To generate a correct call of a procedure or function, the compiler must know (a) its name, and (b) the kinds and types of its parameters; if it is a function, it must also know (c) the result type. Note that it does not need the names of the parameters, though these are of course required when processing the body, and at this time also the name of the result variable of a function if this has been specified.

The signature of a procedure or function is a combination of these requirements, which is in fact the information in the heading (see procedures and functions). In the most common situation, the heading is immediately followed by the body of the declaration, but there are other places in which the heading appears separately: forward-declared procedures, exported procedures, and methods. The signature then serves to ensure that calls match the definition. For further details see Procedures and LRM 8.1.4.

Defining data

This section introduces some basic concepts, then lists the predefined (built-in) data types and describes how to define your own data layouts, and finally shows how to introduce variables into your programs.

The concept of Type

Pascal is what is known as a “strongly typed” language. Every item of data possesses a type, which may be one of the predefined types or a new type which you define yourself to describe the data in your program (so-called new types).   The type determines which operations can be performed on the item, and what effect the operations have. Because all data within the computer is ultimately composed of binary groups, there would be plenty of scope in an untested program for attempting nonsensical operations which produced bogus results that were in turn passed to other processes. The task of unravelling such tangles can be enormous, and while some can still arise, strong typing does help greatly by trapping many potential errors before execution of the program is even attempted.

The classic Pascal standard for the most part only employed types that could be defined at compile time; such aspects as array sizes and string lengths had to be fixed (though see Conformant array parameters). In Extended Pascal, you can introduce schema types which are families of related types from which individuals can be selected at run time. An example might be a family of one-dimensional arrays with lower and upper index bounds supplied either as compile-time or as run-time values.

Like programmer-defined schemas, string and widestring types are defined to have a capacity that can be chosen either at compile time or at run time. Formal string parameters can be specified that adapt to the capacity of the actual parameter at each call.

In Extended Pascal, you can also associate an initial value (more properly known as an initial state) with a type. When any variable, array element or record field that possesses the type is created, the initial value is assigned to it, unless overridden in the declaration of the individual item.

Type compatibility

Pascal uses the concept of type to control associations within a program, and help you to eliminate mistakes at an early stage. There are two kinds of situation in which it applies. The first is concerned to ensure that for example like is compared with like, and that an index value matches the type of the original index definition. In these situations, the two types must be what is known as compatible. This is a symmetrical relationship between two types.

The second situation covers the assignment of a value to a destination such as a variable, element or field, avoiding for instance a day of the week such as Monday being supplied where a day in the month was intended. The assignment may take the form of an assignment statement, or it may be an equivalent such as the passing of a value parameter. The type of the value and that of the destination must be what is called assignment compatible.

Compatible types

This is a symmetrical relationship; if type t1 is compatible with type t2 then t2 is compatible with t1. The two types are compatible if at least one of the following is true:

1. t1 and t2 are the same type
2. t1 is a subrange of t2, or vice versa, or both
   are subranges of the same host type
3. both are string or character types
4. one is a pointer type and the other is type ptr
5. both are set types with the same base type and
   neither or both is packed
6. both are class types.

A particular consequence of 2 is that subranges of integer such as shortint are compatible with integer and with each other. See Subrange types.

In 3, widestring and conventional string types are separately compatible, but not with each other.

In connection with 4, type ptr is not part of standard Pascal. However the constant NIL is, and is compatible with all pointer types. Two pointers can be compared if (by 1) they have the same type, or (by 4) one is NIL or has type ptr. See Pointer types.

Note that arithmetic operations have their own rules. The operands may be compatible, but it is not a requirement.

Assignment compatible types

Assignment compatibility concerns the values that it is sensible to store in a destination, for instance a variable. If tl is the type of the left-hand-side of an assignment and tr is the type of the right-hand-side, then tr is said to be assignment compatible with tl if at least one of the following is true:

1. tl and tr are the same type, and it is an assignable type (see below)
2. tl and tr are compatible ordinal types, string types, or pointer types (see below)
3. tl is real type and tr is integer
4. tl is shortreal and tr is real or integer
5. tl is complex and tr is real or integer
6. tl and tr are compatible set types
7. both are class types and tl is the same as or an ancestor of tr.

An assignable type (see 1) is a type which is not a file type, nor a structured type containing a file.

In 2, the standard specifies some additional restrictions which can usually be checked only when the program is run; they would often be regarded as "run time errors". Essentially, they are overflow conditions such as assigning a value to a string variable whose capacity is insufficient. In this implementation you can request run-time checks to trap such mistakes, otherwise the value is simply truncated.

A widestring or a conventional string expression can be assigned to a widestring variable.

Values, variables and constants

The actions of a program work with values. For example, an arithmetic add operation that takes the value 12 and the value 20 will produce the value 32. When you write 20 in your program, you are introducing a constant whose value is always 20. A variable on the other hand is a container for a value; you can put a value into the container and get it back again later. If you put the value 12 into a variable, and then define an operation to add the contents of the variable and the constant 20, the result will be the value 32.

A value that is the result of an operation can be treated in one of two main ways: it can be used immediately, or it can be put into a variable for use later. An example of immediate use of the result of adding 12 and 20 might be to multiply it by another value; or it may be needed as an input to another process (a parameter). Occasionally, you may simply throw away the result, but that would only be sensible if the operation had some additional effect.

Many of the values used in programs are numeric, but they can be of other kinds, or to use the proper term other types. You will probably often introduce character string values into your programs, for instance `'Hello world!'` is a constant whose value is a character string. You can introduce a variable to hold character string values, put a value into it, and get it back later, just as with a numeric variable. A character string is however different from a numeric value in having an internal structure, namely a sequence of characters, and you can refer to individual characters or groups of characters as well as to the complete value if you wish.

The same ideas apply to other structures. If you introduce a new composite data structure that is convenient to work with, you can introduce variables to hold these structures, and in many cases you can also define constant values. Extended Pascal tries to provide uniform facilities, and to avoid exceptions, but constants of some data types (such as files) would not be sensible.

Pascal programs use names, or more correctly identifiers, to refer to many of the things that make up programs. A variable always has a name, and a constant may have one. It may be helpful to give a name to a constant when it shows its purpose (such as `title` or `MaxSize`), or when the same constant is used in several places.

See also The concept of type and Identifiers.

Initial values

An initial value, more correctly called an initial state, can be associated with a type, or can accompany the declaration of a variable. The initial value in a variable declaration overrides any that may be associated with the type of the variable. An initial value is introduced by the word symbol `VALUE`, and must be a constant or constant expression.

For example:
```
 TYPE  intz = integer VALUE 0;
     ThreeCols = (red, green, blue) VALUE red;
  VAR   count: intz;
     index: integer VALUE 1;
     hue: ThreeCols VALUE blue;
```
Here, the variable `count` will be initialised to 0, `index` to 1, and `hue` to blue (overriding the default associated with the type `ThreeCols`).

The initial state is established when a variable is created. For an outer-level variable, this is before the program starts; for local variables of procedures, it is when the procedure is activated (so they are initialised at every call). Variables in the heap are created by the procedure new.

Initial values can be specified for strings, arrays and records, as well as for simple types such as those above. The value for an array, record or set is given by a constructor containing only constants; alternatively, the fields in a record definition can include individual values. Here are a few more examples:
```
  TYPE  string10 = string(10) VALUE '';
        recptr = ^trec VALUE NIL;
        trec = RECORD
                 next: recptr;
                 a,b: intz;
                 c: char VALUE '*';
               END;
  VAR   recp: recptr;
```
Any variable of type `string10` will be created as an empty string. The variable `recp` will initially contain NIL. After `new(recp)`, it will point to a record with field `next` set to NIL, fields `a` and `b` set to zero, and field `c` set to asterisk.

Predefined types

The following predefined types are provided:

BindingType Boolean byte char complex DaysOfWeek

FileNameType integer real shortint shortreal shortstring

string text TimeStamp wchar widestring word

integer

Kind: Predefined type (all levels)

Values of **integer** type are whole numbers; the largest integer is given by the predefined constant **maxint**, which in this implementation is defined as 214748367. Values in the range -maxint to +maxint are correctly handled. The particular case of -maxint-1 (hexadecimal 80000000) is not well-behaved, for instance it overflows when negated, and is sometimes used to represent "undefined". (LRM 6.1.1.1.1)

Variables of type integer occupy 4 bytes (32 bits). Subranges of integer, such as word, byte, or suitable programmer-defined subranges, take less space in PACKED arrays or records.

word

Kind: Predefined type

The type **word** is predefined as a subrange of integer accommodating values from 0 to 65535, that is, unsigned 16-bit whole numbers. It is provided for convenience, and has no special properties beyond those which apply to programmer-defined subranges. (LRM 6.1.1.1.1 and 6.1.1.3)

Components of packed arrays or records declared as word type are allocated two bytes, and are automatically extended when involved in calculations. Individual variables, or components of non-packed structures, are in most cases allocated four bytes for efficiency reasons, but are assumed to hold values within the defined subrange. Assignment range checks can be applied.

shortint

Kind: Predefined type

The type **shortint** is predefined as a subrange of integer accommodating values from minus 32768 to 32767, that is, signed 16-bit whole numbers. It is provided for convenience, and has no special properties beyond those which apply to programmer-defined subranges. (LRM 6.1.1.1.1 and 6.1.1.3)

Components of packed arrays or records declared as shortint are allocated two bytes, and are automatically extended when involved in calculations. Individual variables, or components of non-packed structures, are in most cases allocated four bytes for efficiency reasons, but are assumed to hold values within the defined subrange. Assignment range checks can be applied.

byte

Kind: Predefined type

The type **byte** is predefined as a subrange of integer accommodating values from 0 to 255, that is, unsigned 8-bit whole numbers. It is provided for convenience, and has no special properties beyond those which apply to programmer-defined subranges. (LRM 6.1.1.1.1 and 6.1.1.3)

Components of packed arrays or records declared as type byte are allocated one byte of storage, and are automatically extended when involved in calculations. Individual variables, or components of non-packed structures, are in most cases allocated four bytes for efficiency reasons, but are assumed to hold values within the defined subrange. Assignment range checks can be applied.

real

Kind: Predefined type (all levels)

Values of type **real** are signed floating-point quantities, with magnitude in the range from **minreal** to **maxreal**. The predefined constants minreal and maxreal in this implementation have values which are approximately 5E-324 and 1.8E+308 (see below). Real values are held internally to a precision equivalent to slightly less than 16 decimal digits.

Variables of type real occupy 8 bytes of storage in the format employed by the numeric processor. The related predefined constants are approximated by:

```
    maxreal     1.8E+308
    minreal     5.0E-324
    minrealn    2.2E-308
    epsreal     2.2E-16
```

(Here, minrealn is the smallest normalised real and minreal is the value below which underflow is signalled.)

shortreal

Kind: Predefined type

Variables (and more particularly array elements) of type **shortreal** can be used to economise on storage when the range and/or accuracy of type real are not required. They can accommodate floating-point values with a range of approximately 1.1E-38 to 3.4E+38, and held to a precision equivalent to slightly more than seven decimal digits. They occupy four bytes of storage rather than eight.

All computations are carried out with a range and precision greater than that of type real, and a reference to a shortreal variable automatically extends its value to the computational format.

complex

Kind: Predefined type (EP standard)

Values of **complex** type are complex numbers; functions are provided which produce a complex value from a pair of real values, or yield the constituent parts, using either Cartesian or polar notation. The standard does not require any particular internal representation, but this implementation uses Cartesian form, with the real and imaginary parts each being of type real.

You introduce complex constants as constant expressions which involve one of the functions cmplx or polar, and you display complex values by separating the real and imaginary parts (functions re and im), or the polar components (abs and arg), and showing them as two real values. (LRM 6.1.1.1.3)

Boolean

Kind: Predefined type (all levels)

Values of **Boolean** type are either false or true; there are predefined constants **false** and **true** whose ordinal values are 0 and 1. Comparison operations return a Boolean result, and these and other Boolean values can be combined using the operators AND and OR. A Boolean value is required in IF, WHILE and REPEAT statements to decide the action of the program, for example:

```
  WHILE (j < 10) AND (k <> 0) DO ...
```

Boolean variables occupy one byte of storage.

char

Kind: Predefined type (all levels)

Values of type **char** are characters; the constant **maxchar** is the value of type char with the greatest ordinal value. In this implementation, characters are 8 bits, having ordinal values in the range 0 to 255 (so that ord(maxchar) = 255). The optional checks for undefined references use maxchar itself as "undefined".

The function ord can be used to obtain the ordinal value of a character, and function chr to obtain the character corresponding to an ordinal or a Unicode character. You can define fixed strings and variable strings of characters with their own properties.

Programs in this implementation may use either the ANSI or OEM character set, as described in 1.10 of LRM.

wchar

Kind: Predefined type

Values of type **wchar** are "wide" 16-bit Unicode characters. The function ord can be used to obtain the ordinal value of a wide character, function wchr to obtain a wide character from an ordinal or an 8-bit character, and function chr to obtain the 8-bit character corresponding to an ordinal or a Unicode character. You can define a string of Unicode characters as type widestring.

text

Kind: Predefined type (all levels)

A variable of type **text** is a textfile, which can be associated with an external file and then used to access that file. There are procedures and functions to make the initial association, to prepare for input or output, and to read or write. (See for instance OpenRead, OpenWrite, readln, writeln.)   Standard input, standard output, and standard error are predefined textfiles that are prepared for use by the run-time system if their names are listed in the program parameters, and can be read or written using the same procedures.

Note that by default the type text is not bindable, and if a variable is to be associated with a named external file you must specify `BINDABLE text`.

DaysOfWeek

Kind: Predefined type

This is an enumerated type defined by:
```
  DaysOfWeek = (Sunday,Monday,Tuesday,Wednesday,Thursday,
          Friday,Saturday)
```
It is used in the definition of TimeStamp, but is not limited to that application.

TimeStamp

Kind: Predefined type (EP standard)

This type is defined as follows.
```
  TimeStamp = PACKED RECORD
          DateValid,
          TimeValid: Boolean;
          Year: integer;
          Month: 1..12;
          Day:   1..31;
          Hour:  0..23;
          Minute:  0..59;
          Second:  0..59;
          Millisec:  0..999;
          WeekDay: DaysOfWeek;
           END;
```
The fields Millisec and WeekDay are not part of the standard definition. (LRM 6.1.2.2)

TimeStamp is used by procedure GetTimeStamp and functions date and time. See also DaysOfWeek.

BindingType

Kind: Predefined type (EP standard)

This type is defined as follows.
```
  BindingType = PACKED RECORD
             name: FileNameType;
             bound: Boolean;
             existing,
             readonly,
             exclusive: Boolean;
           END;
```
The fields existing, readonly and exclusive are not part of the standard definition. They are provided in order to give some control over file selection without resorting to implementation-specific values. (LRM 6.1.2.2)

BindingType is used by the procedure bind and returned by the function binding. See also FileNameType.

FileNameType

Kind: Predefined type

This type is defined as a string which can accommodate the longest normal file pathname; in this implementation it is string(260). (MAX_PATH)

string

Kind: Predefined type (EP standard)

The **string** types are a family of types whose common property is that they are strings of characters. When declaring a variable, or field of a record, you select a member of the family as the type of the variable or field by specifying a **capacity**, that is, the longest string you wish to accommodate, for example `string(50)`. Often, the capacity will be a constant, but a variable such as a parameter can be specified when defining the capacity of a local variable. The essential point is that it must be a value that is defined by the time the string is created.

A formal parameter can be declared to be a **string** without a specified capacity. In such cases, the capacity is taken from the actual parameter, and can be different at each call. A `VAR` formal parameter, for instance, adopts the capacity of the variable that is the actual parameter. If the procedure `GetNextLine` is defined as

```
  PROCEDURE GetNextLine (VAR line: string)
```

then within the procedure, the reference `line.capacity` will produce the capacity of the actual parameter specified by the caller, and suitable action can be taken if it is liable to be exceeded. You can also declare a pointer type as **^string**, and you then supply a capacity in the call of new. Again, it can be different each time.

A reference to a string variable, field or parameter yields a value of the general string type that can be included in string expressions; it can be concatenated with other string or character values, and can be passed to procedures or functions that require a string value. String concatenation is specified using the + operator. Substrings can be referenced with a **[j..k]** notation, and may be either source or destination. For more details on declaring and using strings see LRM 6.1.2.5 and the String handling procedures and functions.

shortstring

Kind: Predefined type

Like string types, the **shortstring** types are a family with the common property of being strings of 8-bit characters. You can use them in the same way as the default strings, but they are provided mainly to give continuity with older programs that contain assumptions about the layout of strings in memory. For more details refer to LRM 6.1.2.5.

widestring

Kind: Predefined type

Like string types, the **widestring** types are a family with the common property of being strings; however, in this case they are strings of "wide" (16-bit) Unicode characters rather than 8-bit characters. You can declare variables and fields in the same way as with the standard string type, by supplying a capacity, for example `widestring(99)`. You can also use the unqualified name as the type of a VAR formal parameter, or the domain type of a pointer.

Because each 8-bit character has an equivalent Unicode character, conventional string values can be assigned to widestring destinations (variables, fields, etc) and conversion can be supplied automatically. Comparisons between widestring variables, or widestrings and conventional string values, can also be performed by the functions eq and ne; a conventional string is automatically converted, and the two Unicode strings are compared. Conversion in the other direction is performed by the function WideToStr, and may involve substituting a default (usually '?') for Unicode characters with no 8-bit equivalent.

In other respects there are differences between the use of widestrings and conventional strings. Concatenation cannot be performed using an operator, though the insert procedure is available and can often serve a similar purpose. For more details refer to LRM 6.1.2.6.

See also:

wchar chr wchr delete insert length setlength

IsAlpha IsDigit LowerCase UpperCase

Programmer-defined types

As well as using the predefined types such as **integer** and **char**, you can define and use new types of your own that describe the data in your program. Indeed, defining your data is an important aspect of software development, and deserves attention just as much as coding the algorithms.

The definition of a new type can be treated in one of two ways. It can be given a name, which is equated to the definition in a type definition part (see below), or it can be written where it is needed as the type of a variable, array element or field. There are some situations in which a type name is required, for instance any formal parameter of a procedure or function must have a named type; if you do not name a new type you cannot use it for these purposes. It is therefore usually a good idea to name data types, and in any case it will help to make your source text clear and meaningful.

A new type can be defined simply as a synonym for a predefined or previously-defined type, and this can be more useful than might at first appear. One important aspect of new types that was introduced in the Extended Pascal standard is the possibility of attaching a default initial value to a type. (You can override the default subsequently if you wish.) So for instance, you could define `boolf` thus:

```
  TYPE  boolf = Boolean VALUE false;
```

The new type has the attributes of Boolean, and in addition has the associated default initial state **false**.

Here is an example of a type definition part that introduces an array and a string:

```
  TYPE  vector = ARRAY [1..100] OF real;
     nametype = string(50);
```

The array type with the name `vector` has 100 real elements, and the string type called `nametype` has a capacity of 50 characters.

There are a several kinds of new type you can introduce:

Enumerated types	Subrange types	
Array types	Record types	
Set types	File types	
Pointer types	Schema types	Class types

The ways in which types are used is the subject of

Type denoters

Enumerated types

An enumerated type introduces an ordered set of values by listing a name for each value. For example, you could write:

```
  TYPE rainbow =
       (violet,indigo,blue,green,yellow,orange,red);
```

The names in this list have two main properties. The first is that they are all associated with the type; if you declare a variable of type rainbow, the only values it can contain are the names in the list. The second property is that they are ordered, in the sense that blue (for instance) follows indigo and precedes green.

An enumerated type is an ordinal type, that is to say, the values of the type map onto a succession of numbers. The first value (violet in the example) has ordinal value 0, the next (indigo) has ordinal value 1, and so on. You cannot perform arithmetic on the enumerated values directly, but you can compare them, and there are functions that allow you to work with them. In the rainbow enumeration, `(blue > indigo)` is true, and so is `(blue < red)`. The function pred gives the previous value, so that `pred(blue)` is indigo, and succ gives the next - successor - value, so that `succ(yellow)` is orange. You can also skip values; `succ(blue,3)` is also orange. The function ord gives the ordinal corresponding to an enumerated value.

Enumerated types are an important aid to security in large programs. Using them helps to trap mistakes such as giving the day of the week instead of the day of the month. The compiler can reject such cases, and you avoid finding them (or even, not finding them) while testing your program.

The predefined type DaysOfWeek is an enumerated type that you are free to use.

Subrange types

A subrange, as the name suggests, defines part of the range of an ordinal type, known as the host type. Commonly, subranges have integer as their host type, but you can define subranges of other ordinal types such char or enumerated types. The definition specifies the lower and upper bounds separated by the ".." symbol.

Examples of subrange types:
```
  0..9               ( host type integer )
  '0'..'9'           ( host type char )
  Monday..Friday     ( host type DaysOfWeek )
```
The predefined types byte, word and shortint are also examples of subranges whose host type is integer.

When you refer to a variable or element having a subrange type, its value is treated as being of the host type. For subranges of integer, this means that the value is expanded (if necessary) to 4-byte width before taking part in a calculation or being passed as a parameter. The code generation may assume that the value lies within the subrange. When a value is assigned to a variable or element having a subrange type, you can request the compiler to insert a check to ensure that it is indeed within the subrange bounds.

Subranges are often defined with constant bounds (for instance 0..9 above), but you can also have bounds which are defined at run time. Most often, such usage is associated with defining arrays whose size is not known at compile time, as described in Schematic arrays, but a subrange with variable bounds can have other uses such as in indexed (direct access) files.

Array types

An array type defines a number of elements of the same type, individual elements being selected by an index. You can define arrays of simple types, such as integer, char or real; or the elements can be records, strings, or other arrays. The index is often numeric, but the elements can be chosen by enumeration values, characters, or any ordinal type. In the definition of the type, the index type determines the number of elements in the array.

The following examples show a few of the possibilities:

```
  TYPE  subr = 1..100;         (defines a subrange)
     iarray = ARRAY [-10..+10] OF integer;
     vector = ARRAY [subr] OF real;
     spectrum = ARRAY [rainbow] OF real;
```

Here, the array iarr has 21 integer elements, and vect has 100 real elements. The type rainbow was an example of an enumerated type, and spectrum is an array with one real element for each of the enumerated values violet to red; it might be used to hold wavelength values, say.

You can define arrays of other element types, and multi-dimensioned arrays, for instance:

```
  TYPE  str10 = string(10);    (defines a string type)
     starray = ARRAY [DaysOfWeek] OF str10;
     dtarray = ARRAY [0..4] OF TimeStamp;
     twodims = ARRAY [2..5] OF vector;
     matrix = ARRAY [2..5,subr] OF real;
```

The types DaysOfWeek and TimeStamp are predefined. The array types `twodims` and `matrix` are alternative ways of defining similar two-dimensional arrays; the matrix type could also have been defined as `ARRAY[2..5] OF ARRAY[subr] OF real`.

The types `twodims` and `matrix` illustrate a general point: while they are similar, they are not "the same". Pascal allows you to do various things with structures of the same type: you can assign one to another, or pass one to a formal parameter of the same type. Because `twodims` and `matrix` are in fact array types with the same shape, you could assign a variable of one type to a variable of the other element-by-element, but you cannot perform whole-array operations between them. See VAR parameters.

When you have an array of small elements, and the size of the array is more significant than the small amount of extra code that may be needed to access the individual elements, you can define the array to be `PACKED`. For example:

```
  TYPE  buffer = PACKED ARRAY [0..9999] OF byte;
```

In this implementation, such an array will occupy 10000 bytes, while the non-packed form would occupy 40000 bytes. Packing in fact mainly affects element types which are subranges of integer, such as byte and word; char or Boolean, or enumerated types, do not benefit, nor do the larger types such as integer or real.

You can describe a complete array value (for instance a constant or an initial value) by means of an array constructor.

Array constructors
An array constructor defines the value of a whole array; when all the ingredients are constant, the value can be named as a constant or used as an initial value. Suppose that you have a type defined as `ARRAY [-10..+10] OF integer`, then a constructor specifies indexes and the values that the corresponding elements should have, for example:

```
    [ -6,-4,+4: 10; -5,-3,+3: 11; -2..+2: 12;
      OTHERWISE 5 ]
```

Besides specifying that particular elements should have the values 10, 11 and 12, this gives the default value 5 to all other elements. An array constructor must give a value to every element, and the OTHERWISE clause allows you to ensure this even when the range of index values is not defined until run time (see Schema types). Indeed, a useful array constructor is `[OTHERWISE 0]`.

When a constructor is used within a type definition as the initial value, its type is implicit, but in other situations it must be specified. For example, if the above type had been defined with the name `iarray`, you could introduce a constant:

```
  CONST iarcons = iarray [-10..-5: 10; 5..10: -10;
                          OTHERWISE 0 ];
```

You could then supply the constant as a value parameter when the formal has type iarray (see Value parameters), and there is another interesting usage. You can refer to an element of an array constant using a variable index, for instance `iarcons[i]`, or in order to select one string from an array of strings.

Record types

A record is a structure made up of named fields, which can include strings, arrays or other records as well as simple types such as integers and characters. You access individual fields by name, typically thus: `recordvar.fieldname`. The definition of a record type is introduced by the word symbol `RECORD` and terminated by the word symbol `END`. For example:

```
  TYPE  vehicle = RECORD
              wheels: 2..12;
              weight: shortreal;
              colour: rainbow;
            END;
  VAR   trucks: ARRAY [1..20] OF vehicle;
```

You can refer to the weight of the seventh vehicle in the array as `trucks[7].weight`.

A record can be defined to have a fixed part, similar to the above, followed by a variant part. The idea is that you can define a number of alternative layouts (variants) which are all nevertheless the same type. See Variant records.

You can define a record type to be packed by placing the word symbol `PACKED` before `RECORD`. Usually, the reason for doing this is to indicate to the compiler that you attach more importance to reducing the size of records than to any extra code that may be needed to address them. Another possible reason is to have records whose internal layout is the same as that produced by another implementation; in particular, they will match the layout from 16-bit Extended Pascal.

Variant records

A variant record consists of a fixed part (which may have no fields in it) and a variant part. The following example defines a record type with three variants:

```
  TYPE  shapes = (triangle, square, circle);
     vntrec = RECORD
             OffsetX, OffsetY: real;
             CASE shape: shapes OF
               triangle: (side1, side2,
                    side3: real);
               square:   (side: real);
               circle:   (radius: real);
           END;
```

As will be seen, the fixed part consists of the two fields OffsetX and OffsetY, the variant part is introduced by `CASE...OF`, and the list of fields for each variant is enclosed in parentheses. The tagtype, in this case `shapes`, specifies the basis on which the variants are defined, and the possible values are listed. The tag, in this example `shape`, determines the currently selected variant, the "active" variant. Including the tag value in the definition is optional, but is generally recommended.

All the possible values of the tagtype must be accounted for in the list of variants, but not necessarily separately. You can have multiple tag values against one list of fields, thus: `tag1,tag2,tag3: (...);` or `tag5..tag9: (...);`. Also, you can use OTHERWISE at the end to account for any remaining values: `OTHERWISE (...);`. (For comparison, see CASE statement.)

The Pascal standard does not demand that variants should be allocated overlapping storage, but almost all implementations do so in almost all cases. If you use the procedure new to allocate space in the heap for a variant record, you can optionally supply a tag value, and would get just the space needed for the corresponding variant. In our example above, the record describing a square or a circle requires less space than one describing a triangle. (See LRM 9.4.2 for more detailed rules concerning such usage.)

This implementation includes an optional check on variant usage, explained in the User Manual. See also schematic records.

Record constructors
Just as an array constructor defines the value of an array, so a record constructor defines the value of a whole record; when all the ingredients are constant, the value can be named as a constant or used as an initial value.

To illustrate the format of a record constructor, take the type `vehicle` from the description of record types. A constructor names each field, and gives its value:

```
    [wheels: 8; weight: 13.75; colour: red]
```

Set types

A set type describes a set of values of a base type; they could be for example those members of the **char** type which are vowels, or the days from DaysOfWeek on which an extra discount is given. The base type must be an ordinal type.

A set type is defined thus:

```
  TYPE  stype = SET OF basetype;
```

for example

```
  TYPE  charset = SET OF char;
```

A variable of type charset can be used to define characters having a common property, such as being printable.

Set values of compatible base types can be combined, using the conventional operators, which in this situation take on special meanings. The + operator gives the union of the two operands, – gives the difference, and * the intersection; comparisons can also be performed, again with special meanings. For more details see Set operations.

A set value can be specified by means of a **set constructor**, which shows the members to be included, contained between square brackets, for instance `['a','e','i','o','u']`. When the members are all constants, as here, the set value is a constant, and can specify an initial state. A constructor within an executable statement can also specify a set which includes members defined by variables.

File types

The predefined file type text is one of the common means by which a program can communicate with its environment. Textfiles are composed of characters arranged into lines. You can define files with other component types, such as records, arrays, or indeed simpler types such as integers. The only requirement is that it must be an "assignable type" (see under Assignment-compatible types). Non-text files are often referred to collectively as binary files.

It is necessary to be clear here about terminology. A Pascal variable having a file type is commonly referred to as a "file", but so also is an external file to which the variable may be connected. The term "file variable" will be used when there may be doubt as to which is intended.

File variables can be defined to access an external file either sequentially or randomly. The form of definition for sequential access is

```
  TYPE  fseq = FILE OF ComponentType;
```

and for random (or direct) access is

```
  TYPE  fdir = FILE [IndexType] OF ComponentType;
```

The same external file can be connected at one time to a variable of a sequential type and at another time to a variable of a direct-access type. Using the sequential file variable, the components of the external file must be processed sequentially; with the direct-access file variable they can be processed randomly or sequentially. The index type in the definition of a direct-access file type must be able to cover the number of components in the external file, and can exceed that number; for instance `[1..maxint]` can be used.

File variables are connected to external files, and components read and written, by means of predeclared procedures and functions. For making connection see
  OpenRead, OpenWrite, bind, reset and rewrite. To read, write or modify components, see
  read, write and update. For accessing components of direct-access files, see
  SeekRead, SeekWrite and SeekUpdate.
Other file-handling routines are listed in the File handling section.

Pointer types

An executing Pascal program includes an area of memory called the heap which is available for dynamic allocation and disposal. Variables can be created in the heap, and are then accessed by pointers. When such a variable is no longer needed, it can be "disposed" and the space it occupied recovered.

The pointers are themselves variables, and it is important to keep in mind the difference between the pointer and the variable to which it points (the identified variable, in the standard definition). A pointer is a variable having a pointer type, and each pointer type is associated with a particular domain type, that is to say, the type of the variable in the heap to which it points.

On exit from a procedure or function, any local variables disappear. By contrast, any variables which it may have created in the heap continue to exist. Provided that the pointers to these variables have been passed out of the procedure, via parameters or by copying to longer-lived environments, the identified variables can still be used.

A pointer type is defined by association with a domain type, thus:

```
  TYPE  pdomain = ^domain;
```

It is helpful to give the pointer type a name that shows the connection with the domain type, and prefixing a P to the name of the domain type is a conventional way to do this. In this definition, "domain" is an illustrative name; you use the name of the type that you wish the pointer to address. A pointer to the type **vehicle** shown in Record types, for example, might be defined by

```
  TYPE  pvehicle = ^vehicle;
```

If a pointer variable of type pvehicle, **pv** say, is declared, you create a vehicle record in the heap by calling the procedure new with `pv` as parameter. The operation leaves `pv` addressing the new record, and you access the record with the notation `pv^`, addressing individual fields by `pv^.weight` for instance.

The definition of pointer types has one special property, which is an exception to the general rule that names must be defined before they can be used. In order to be able to construct linked lists of records in the heap (which is a very handy technique for certain purposes) it is necessary to include a pointer as a field in the record which links to the next in the list. Such a pointer must have a type whose domain type is the type of the record, that is, it requires a mutual reference; under the normal rule, this would be impossible. To allow it, when defining a pointer type you can include a domain name which is a forward reference, for example:

```
  TYPE  prec = ^rec;
     rec   = RECORD
           next: prec;
           (* other fields *)
            END;
```

You can then scan through the list by next^, or by copying next to another variable v of the same pointer type and referring to v^. The definitions of prec and rec need not be adjacent as in that example, but must be in the same "type definition part" introduced by the word TYPE.

Schema types

A schema type is a family of related types, together with one or more parameters or discriminants. You can select a member of the family by supplying actual values for the discriminants, somewhat as you supply actual parameters in a procedure call. Schema types allow you to define families of subranges, arrays and records; and because the selection of an individual member can take place at run time, it can depend upon a run-time value or values rather than being fixed at compile time.

There is therefore a two-stage process: first, you define the "model" or "prototype", together with factors which will decide the selection of specific members; then, you can pick individual types to suit particular purposes. The factors might be for example the index bounds of an array, which will in turn decide the size.

The string type is technically a predefined schema, with one discriminant called **capacity**. You can select individual string types by specifying a value for the capacity. The shortstring and widestring types are similar.

A type selected from a schema family is known as a schematic type, and is one form of type denoter. Two schematic types are considered "the same" (and are therefore compatible) if they were produced from the same schema with the same set of discriminant values. The schema itself is not a valid type denoter, but can be used in two situations: as the type of a formal parameter, and as the domain type of a pointer. The schema in effect adapts to events at run time, for example any string-type value can be passed to a formal parameter of type **string**.

The following sections include examples of schema definitions and usage:

Schematic subranges

Schematic arrays

Schematic records

For a more detailed general description of schema types see section 6.2 in the Language Reference Manual.

Schematic subranges

You can define a schema which is a family of subrange types like this:

```
  TYPE  subschema(lower,upper: integer) = lower..upper;
```

The types produced from this schema are subranges of integer; alternatively you might have a schema which is a family of subranges of char:

```
  TYPE  charsubs(first,last: char) = first..last;
```

In these definitions, lower, upper, first and last are the discriminants, and you produce individual types from the schema by substituting values for these discriminants, for example:

```
  TYPE  TenToTwenty = subschema(10,20);
```

As is quite often the case, the discriminants appear as the bounds of the subrange, and for this reason must obey the rules (be of compatible types, and the value substituted for lower must not exceed that for upper) when a type is produced.

Discriminants will normally appear in the definition of the schema, but not necessarily in the simple form shown above. It might be convenient to have a schema like this:

```
  TYPE  basecount(base: integer; count: word) =
                 base .. base+count-1;
```

A subrange is defined by two expressions, and you can take advantage of this, as here; however, the expressions can only refer to discriminants and constants. You can produce a type from the schema wherever a type denoter is valid, such as:

```
  VAR   sv: basecount(0,100);
```

The variable `sv` has a subrange type with bounds 0 to 99 (0+100-1).

Schematic subranges can be incorporated into other schemas. You might use one as the index type for a family of arrays, allowing the subrange schema to be used independently elsewhere in your program.

Schematic arrays

One of the useful applications of schema types is to describe arrays which can adapt to different circumstances at run-time. The schema `vec` defined by:

```
  TYPE  vec(limit: integer) = ARRAY[1..limit] OF integer;
```

can be used as the type of a parameter like this:

```
  FUNCTION  sum (VAR fvec: vec) = ans: real;
    VAR   n: integer;
    BEGIN
      ans := 0;
      FOR n := 1 TO fvec.limit DO
       ans := ans + fvec[n];
    END (*sum*);
```

Within the function, the code can refer to the upper bound of the index as `fvec.limit`. If elsewhere in the program there are variables declared as:

```
  VAR   factors: vec(10);
     results: vec(32);
```

they can be passed to the function, say like this:

```
     writeln(sum(factors),sum(results));
```

and the function will adjust automatically to the different sizes.

Another possible application of the schema `vec` is to declare a local variable whose size is defined by a parameter. In the procedure which starts:

```
  PROCEDURE proc (size: integer);
    VAR   lvec: vec(size);
```

the variable `lvec` has a type produced from the schema with the discriminant value `size`. Within procedure `proc`, the array `lvec` can be passed to the function `sum`, and the reference within `sum` to `fvec.limit` will return the value of `size`.

The principles described above in terms of a vector can easily be extended to arrays of two or dimensions. The following example is probably rather artificial, but will illustrate some more possibilities.

```
  TYPE  sub(first,last: char) = first..last;
     twodim(low1,upp1,upp2: integer; f,l: char) =
        ARRAY[low1..upp1,1..upp2] OF sub(f,l);
```

The family of types defined by `twodim` are arrays with index bounds derived from the discriminants `low1`, `upp1` and `upp2`. The component type is a subrange of **char** produced from the schema `sub` by substituting the discriminants `f` and `l` of `twodim` for the discriminants `first` and `last` of `sub`.

It is worth noticing at this point some differences between schema parameters as described above and conformant array parameters. A function equivalent to `sum` could be declared with a conformant array parameter, and would adapt to actual parameters of different sizes; also, the actual parameters need not all be produced from the same schema, but can be any arrays that "conform" to the pattern of a one-dimensional array of reals. However, they must already exist; the conformant array method does not provide any means of creating new arrays, such as `lvec` in the procedure `proc`.

Schematic records

The description of record types introduced the form of <u>variant record</u> in which the active variant is selected by a tag value. The application of schemas to records is based on using a discriminant as the variant selector. Here is an example:

```
  TYPE  recschema(select: Boolean) =
          RECORD
            f1, f2: anytype;
            CASE select OF
           false: ( ..fields.. );
           true:  ( ..fields.. );
          END;
```

The selector can be any <u>ordinal</u> type, which for practical purposes should have a limited range of values. You produce specific record types from such a schema by supplying an actual discriminant value (in the example, either false or true).

As with other schema types, you can pass records as parameters when the formal parameter has the schema type, and within the procedure or function discover the type of each actual. You can also create a local variable whose type is produced when the procedure or function is entered (see <u>schematic arrays</u> for an example). Another possibility available for all schema types, but probably most useful with records, is to define a pointer having the schema type. Following the example above, you could write something like this:

```
  TYPE  precschema = ^recschema;
  VAR   recptr: precschema;
```

Then in subsequent code you can create a record in the heap having a type produced from the schema, by quoting an actual discriminant in a call of **new**:

```
     new (recptr, false);
```

To remove the record from the heap when you have finished using it, you just call `dispose(recptr)`. To build a linked list of records (of either variant) you would introduce a field (called `link`, say) of type `precschema`.

An essential difference between a schematic record and a conventional variant record is that the type (and hence the variant) of the schematic record is decided when the variable is created, whereas (unless it was created in the heap with specific tag values) the variant in the conventional record can be changed during the life of the record. Because many hard-to-find bugs arise from mistakes in variant usage, the schematic form is more secure, but for some purposes the flexibility of the conventional form may be important.

Ordinal and simple types

The term ordinal types describes a collection of types which have a common property: they are either numeric integers, or they can be mapped onto integers (their "ordinal values"), and are indeed represented internally by these ordinal values. Some predefined types are ordinal types, in particular:

  integer, and integer subranges such as word,
  char (character type).

Other ordinal types are defined within the program:

  enumerated types,
    specified subranges of ordinal types.

In a number of situations, an ordinal type is required, for example as the type of an array index, or the selector in a CASE .. OF.

The so-called simple types comprise ordinal types and the real (floating-point) and complex types.

Type denoters

The title of this topic may sound rather technical, but it is a convenient way of introducing the usage of types, as distinct from their definitions. You give each variable a type when you declare it; similarly, you specify the types of array elements and record fields. These references to types are collectively type denoters.

A type denoter may be, and often is, just the name of a type, such as **integer**, or a name that you have defined as a new type. Alternatively, it may be the definition of a new type as described in programmer defined types, such as an enumeration or a record. Again, it can be a type produced from a schema, in particular a string of a specified capacity produced from the predefined **string** schema, for instance `string(50)`.

The following semi-formal definitions show some of the various uses of type denoters.

```
    TYPE  NewType = <TypeDenoter>;
       VecType = ARRAY [1..n] OF <TypeDenoter>;
       RecType = RECORD
                field1: <TypeDenoter>;
                field2: <TypeDenoter>;
              END;
    VAR   var1,var2: <TypeDenoter>;
```

An identifier can be applied at any point after its definition is complete; because NewType is defined as the name of a type, it would be a valid form of TypeDenoter in any of the later occurrences, such as the types of fields or variables.

A type denoter can also associate the bindable attribute with a type, and provide an initial value. A file variable must have a bindable type if it is to be connected to a named external file, so you might write

```
    VAR   logfile: BINDABLE text;
```

for example, where `BINDABLE text` is a form of type denoter. And in the following, extracted from the description of **initial values**:

```
 TYPE  ThreeCols = (red, green, blue) VALUE red;
 VAR   hue: ThreeCols VALUE blue;
```

both `(red, green, blue) VALUE red` and `ThreeCol VALUE blue` are type denoters.

Variable declarations

You introduce variables into a program in a variable declaration part, giving them each a name and a type. Here is an example:

```
  VAR   j,k: integer;
     title: string(10) VALUE '';
     ch: char;
```

The variables j and k are of type integer, title is a string of capacity 10 which is initialised to empty, and ch is a character. The declaration part is introduced by the word-symbol VAR and can contain one or more declarations each terminated by semicolon. A declaration can name one or more variables of the same type.

A variable declaration part is one of the possible "definitions and declarations" that come at the start of a block. A main program consists of a program-heading and a block; variables declared in this block exist throughout the execution of the program, and are sometimes known as "static" or "outer-level" variables. A procedure or function consists of a procedure-heading or function-heading and a block; variables declared in these blocks exist only while the routine is active, and are known as "local" variables. In Scope there is a description of which variables can be accessed from different parts of a program.

The type which is associated with a variable in its declaration governs the values it can hold and the operations in which it can take part. At the most obvious level, an integer value such as the result of an arithmetic operation with integer inputs can be assigned to (ie. placed in) an integer variable. Up-level arithmetic assignments such as integer value to real variable are also allowed, with conversion supplied automatically. The reverse is a little more difficult, because a real (floating-point) value will often have a fractional part, and also may overflow the limits of the range of integers. There are two functions called trunc and round which take care of these potential difficulties.

In a similar way, integer and string values cannot be directly combined, but the built-in procedure writestr will convert a numeric value to its external representation and place it in a string variable. (Indeed, writestr will build up a string containing multiple values of different types.) Similarly, readstr performs the reverse process.

Accessing variables

To access a variable introduced by a variable declaration, you simply quote its name. The name accesses the whole (entire) variable, which in the case of any simple variable such as an integer or a character is all you need. You can also access whole arrays, strings or records by name, and particularly with strings this will again often be what is required. The notations for accessing elements of arrays by indexing or selecting fields of records are described in Array types and Record types. If you have a variable which is an array of integers, vints say, you can access an element by quoting an index such as vints[i]. Such a reference has the type of the element, in this case integer. Similarly, a record field is selected by name using the "." notation, and the reference has the type of the field. Individual characters from a string can also be accessed by indexing.

To go from a pointer variable to the variable at which it points you use the "^" notation described in Pointer types. When the variable in the heap is a record, you can then select a field by combining the pointer dereference with a field selection, for instance `pv^.weight`.

Just as the definitions of structured types can include other structures, the notations for accessing them may involve a field access followed by an index, or an index followed by a field access. The concept of a variable access is to progress from a base reference to the particular item of interest by a combination of the indexing ([..]), field selection (.) and pointer dereference (^) operations. At each stage the item you have accessed has a type that you can treat as a whole, or (if it is a structured or pointer type) determines what method of further access is appropriate.

Specifying actions

The actions of a program are specified by statements.

- Statements
    - Assignment statement
    - Procedure statement
    - Compound statement
    - GOTO statement
    - Empty statement
    - Constructor statement
    - Destructor statement
    - IF statement
    - CASE statement
    - WHILE statement
    - REPEAT statement
    - FOR statement
    - TRY statement

Statements

The actions (the "algorithm") of a program are defined by statements. In the description of a Main program, the **statement part** was introduced by the symbol BEGIN and terminated by END, and the same structure is used in the body of a procedure or function. The statement part is a sequence of individual statements which are normally obeyed in the order in which they are written, but the order can be modified by means of conditional statements and repetitive statements which are discussed below. Adjacent statements must be separated by a semicolon symbol.

Many statements involve **expressions**. The simplest expressions consist of a constant or a simple reference to a variable; by combining these simple elements with operators such as + or <= you can obtain any desired value. For details see expressions.

The first category of statement consists of "simple" and "compound" statements. These are the statements that are obeyed one after the other in the order in which they are written.

- Assignment statement
- Procedure statement
- Compound statement
- GOTO statement
- Empty statement

The next category are the "conditional" statements that allow you to choose between alternative courses of action.

- IF statement
- CASE statement

The "repetitive" statements allow you to specify that a subsection of the algorithm is to be repeated until an end condition is reached.

- REPEAT statement
- WHILE statement
- FOR statement

(A mistake made by programmers at all levels is to specify an end condition which for some reason is never reached; the program then reports an exception, or alternatively appears to go to sleep and has to be terminated by the user.)

The WITH statement opens up access to the fields of a record, or a schematic variable, and finally there are statements concerned with object-oriented programming and exception handling.

- Constructor statement
- Destructor statement
- TRY statement

Assignment statement

An assignment statement causes a value to be obtained and "assigned" to a variable, array element, or field of a record or object. The destination of the assignment, or left-hand side, is written first, followed by the symbol := and an expression that produces the desired value. For example, if x is an array and r is a record, you might write the following series of assignment statements:

```
     j := count * size;
     x[j] := sin(t);
     r.name := 'April';
```

Those examples assigned to an element of the array x and a field of the record r. In Pascal, you can also assign whole arrays and records (an expression can consist of an array or record), as well as string values involving concatenation and the results of string functions. Some type conversions are supplied automatically: in numeric assignments, from integer to real, and from integer or real to complex; and in character assignments, from an 8-bit character or string to Unicode. In these cases, the destination can always hold the expression value. There is a requirement that the types of the expression and the destination be what is known as assignment compatible, which can generally be checked at compile time. The same rule applies to similar situations, notably parameter passing, and is examined in Type compatibility.

Even when the types of the expression and the destination agree, assignment errors can still occur when the program is executed. A numeric expression may sometimes overflow, or one of its ingredients may not be properly defined; the value of a string expression may be too long for the destination. In this implementation, a floating-point overflow (which is a very unusual event) is always notified, but integer and string overflows are just truncated unless you request the compiler to include checking.

Procedure statement

A procedure statement activates a procedure, which may be one of the predefined procedures such as writeln or bind, a procedure imported from a library, or a procedure that you have defined yourself. If the procedure being called has a parameter list, the statement must supply matching actual parameters.

Examples: writeln ('Hello world!'); delete (sv, 1, 1); GetTimeStamp (ts);

Compound statement

A compound statement is a sequence of statements introduced by the symbol `BEGIN` and terminated by `END`.

Example:

```
     IF ... THEN
       BEGIN
         inx := 1;  writeln ('Next item');
       END;
```

Here, the assignment to `inx` and the writeln are both to be performed if the condition is met. Individual statements in the list are separated by semicolons. The layout is (as always) optional, and in many cases statements would be placed on separate lines. Indenting between the `BEGIN` and `END` gives a clear visual indication of the length of the sequence.

A compound statement forms the statement part which specifies the actions of a program, procedure or function.

GOTO statement

A GOTO statement unconditionally transfers control to a labelled point in the current procedure (a "local" GOTO), or in a textually enclosing procedure or main program ("non-local"). You can use GOTO when you need to leave a repetitive operation before the terminating condition is reached. You must not use a GOTO when the position of the label would involve jumping into a loop or other structure.

Examples: `GOTO 99;` `GOTO next_item;`

The position of the label is defined by writing it, followed by a colon, before the statement which is to be the destination of the GOTO. The identity of the label must appear in a declaration part in the block in which its position is defined.

Empty statement

An empty statement consists of nothing at all; it is nevertheless sometimes convenient, for instance you can label it as the destination of a GOTO when there is no action to be performed.

Constructor statement

In object-oriented programming, a *constructor statement* is used within the implementation of a constructor to activate a constructor defined in a parent class. See also LRM section 13.

Destructor statement

In object-oriented programming, a *destructor statement* is used within the implementation of a destructor to activate a destructor defined in a parent class.

IF statement

There are two forms of IF statement:

IF condition THEN statement
IF condition THEN statement ELSE statement

In both forms the condition is a Boolean expression, and the statements following THEN and ELSE can be compound (BEGIN ... END) to attach a group of actions to one condition.

The statements following THEN and ELSE can also be of other kinds, including IF statements, when some care is needed. The rule in these circumstances is that an ELSE attaches to the nearest IF that does not already have an ELSE. You may sometimes have to put a subsidiary condition by itself inside a BEGIN ... END to avoid saying something you did not intend.

Examples:

```
IF (n < 0) AND (ch = ' ') THEN nextch;
IF (n < 0) THEN
  BEGIN
    IF (ch = ' ') THEN nextch;
  END
ELSE name := name + ch;
```

In the second example, the ELSE belongs to the IF (n < 0) THEN ... , not to the IF (ch = ' ') THEN ... .

If the condition can be evaluated at compile time, a statement that will never be reached is not compiled. Typically, you have a named constant called (say) TestVersion, and use it like this:

```
IF TestVersion THEN writeln('Value of x is: ', x);
```

CASE statement

A CASE statement provides a multi-way choice of actions. The value of an expression is matched to a list of case-constants with each of which is associated a statement, laid out like this:

```
CASE expression OF
  const1:   statementA;
  const2, const3:  statementB;
  const4 .. const5:  statementC;
  OTHERWISE  statements;        optional
END {case}
```

The case-constants must match the type of the expression, which can be any ordinal type such as integer, an enumerated type, or char. They can be constant-expressions of the matching type, and there can be more than one value associated with a statement. (The notation ".." implies a range of values.)

Example:

```
CASE day OF
  Sunday:   writeln ('Gone fishing');
  Monday, Tuesday:  EarlyStart;
  Wednesday .. Friday:  LateFinish;
  Saturday: Shopping;
END {case};
```

The optional OTHERWISE clause (called a "completer") allows you to specify actions to be performed if the expression value does not match any of the case-constants. Without a completer, a non-matching expression raises an exception.

A CASE statement, like an IF statement, provides a means of conditional compilation when the expression can be evaluated at compile time. The selection is made then, and only one of the dependent statements is compiled.

WHILE statement

With a WHILE statement you can obey an action zero or more times. The format is:

```
     WHILE condition DO statement
```

The condition here is a Boolean expression such as `(inx <= len) AND (s[inx] <> ' ')` or `(p <> NIL)`. Control returns each time to test the condition, and so long as it is true the subsidiary statement is obeyed. The latter can be any kind of statement, including a compound statement or even another WHILE.

If the condition is false when the WHILE is first entered, the controlled statement is never executed. You can leave the WHILE loop by means of a GOTO, but this is rarely done, and one of the things the controlled statement must then ensure is that it directly or indirectly modifies the condition so that it eventually fails and the repetition ends.

A form of condition which is useful when processing linked lists is `(p <> NIL) AND_THEN (p^.field < 10)`. The reference to the variable pointed at by p, in this example a record, is only attempted when p is non-NIL.

See also:
  REPEAT statement
  FOR statement

REPEAT statement

With a REPEAT statement you can obey a series of actions one or more times. The format is:

```
     REPEAT statement-sequence UNTIL condition
```

The condition here is a Boolean expression such as `(inx > len)` or `(p = NIL)`. Each time the end of the statement-sequence is reached, the condition is tested, and if it is false control returns to the REPEAT. The statement-sequence is similar to a compound statement, consisting of one or more statements separated by semicolons.

In a WHILE statement, the controlled statement is never executed if the condition is false when the WHILE is first entered, whereas the statement-sequence in a REPEAT loop is always obeyed at least once. You can in principle leave the loop by means of a GOTO, but this is rarely done, and you must otherwise ensure that eventually the condition becomes true and the repetition ends.

See also:
  WHILE statement
  FOR statement

FOR statement

With a FOR statement you can repeat an action a number of times, under the control of a loop count which is maintained automatically. You nominate a variable, which within a procedure or function must be a local variable, as the control variable for the loop. The control variable must have an ordinal type; often it will be an integer or integer subrange such as word, but it may also be char or an enumerated type. Within the loop, the control variable is updated automatically, and you can use it, for instance as an index.

There are two kinds of FOR statement. The one most often used is the sequential form, in which the cotrol variable takes an ascending or descending sequence of values. Another form was introduced in the Extended Pascal standard, in which it takes values defined by means of a set; this is called set-member iteration.

The FOR statement is one form of repetitive statement. See also:
REPEAT statement
WHILE statement

Sequence iteration

In this form of FOR statement, the control variable takes ascending or descending sequential values. The following specifies an ascending sequence:

f1 `FOR v := lower TO upper DO statement`

and this specifies a descending sequence:

```
    FOR v := upper DOWNTO lower DO statement
```

Here, v is the control variable (see the general description of FOR statements), and upper and lower are expressions which specify the range of values it is to take. The controlled statement forms the body of the loop, and can be of any kind; it will often be a compound statement (BEGIN ... END), when several actions are to be carried out at each repetition, but it can alternatively be a single statement, even another FOR statement.

Example:

```
    FOR n := 1 TO size DO buff[n] := 0;
```

In the example, lower is a constant and size is (we assume) a variable, but they can be more complicated expressions when the situation requires; they must however be assignment compatible with the control variable. If the values of lower and upper are equal, the loop is performed once. It is not an error for lower to be greater than upper, but in that case the loop is not entered.

Set-member iteration

In this kind of FOR statement, the control variable takes values defined by a set (see Set types). The format is:

```
    FOR cv IN setvalue DO statement
```

for example

```
    FOR ch IN ['a','e','i','o','u'] DO peek(ch)
```

(where peek is a local procedure).

A DO loop of this kind allows you to express conveniently some non-sequential sets of values. In this implementation the values are in fact selected in ascending order, but it is as well not to rely on this, as there is no such requirement in the standard definition.

WITH statement

The `WITH` statement is part of classic Pascal, and has the format:

```
     WITH record-access DO statement
```

It selects a record, and makes the names of its fields immediately accessible during the contained statement. Returning to the example in record types:

```
  TYPE  vehicle = RECORD
              wheels: 2..12;
              weight: shortreal;
              colour: rainbow;
            END;
  VAR   trucks: ARRAY [1..20] OF vehicle;
```

the following WITH statement allows fields `wheels` and `colour` to be addressed:

```
     WITH trucks[j] DO
       IF (wheels = 2) AND (colour = red) THEN ...
```

Using `WITH` to select the variable is more convenient than repeating the `trucks[j]`, and may well also be more efficient.

In Extended Pascal, a `WITH` statement can also be used to select a variable of a schematic type, and make the names of the schema discriminants immediately accessible.

One thing to beware of is the situation in which a variable in scope has the same name as one of the record fields or discriminants. The names made accessible by `WITH` take precedence, and the variable cannot be addressed while it is in force.

TRY statement

The TRY statement is part of the exception handling feature, which is described in the section Violations, errors and exceptions. It allows you to intercept an exception, and if possible deal with the situation before it is notified to the user of the program. What follows is a description of the commonest usage; there are other possibilities, described in section 14 of the Language Reference Manual. Note that to use the facility, you must first import a supplied interface called `ExceptionHandling`.

The general form of `TRY` statement is

```
TRY  <region>
  <ON-clause>
 [<ON-clause> ...]
END;
```

The `region` of a `TRY` statement is a statement or statement-sequence, which may include procedure calls. After the region is a series of one or more `ON`-clauses, each of which has the form

```
ON exception-name DO statement;
```

The exception names, which are grouped into classes, are listed in

Exception classes.

You can name a specific exception, such as `ExcepInputInteger`, or a group, such as `ExcepFormat` which contains `ExcepInputInteger` together with a number of other specific exceptions related to input/output formatting.

If an exception arises during execution of the region code, control is passed to the first `ON`-clause, and the kind of exception is compared with the exception name in the clause. If the name describes the exception, the related statement is executed, otherwise control passes to the next `ON` clause if there is one. If no clause describes the exception, it is passed up the chain of procedure calls in case an enclosing `TRY` statement can handle the situation. If no matching `ON` clause is found at any level, the user is notified in the usual way.

Expressions

Statements are the means of specifying the actions of a program, but they do not of themselves include any arithmetic or logical capabilities. These are provided by expressions, which take part directly in many kinds of statement, and are also required in other situations such as supplying actual value parameters.

An expression is the means by which you can derive a value (see Values, variables and constants). The simplest expressions consist of a constant, or a reference to a variable, array element or record field; such expressions just produce the value of the item they contain. By combining simple elements with operators such as + or <= you can build up more complicated expressions. Operators are defined which produce arithmetic, Boolean (logical), and string-type values, but not arrays or records; that is, you cannot combine these structured values. For uniformity of definition, however, an expression can consist of a single structured-type value. An assignment statement obtains the value of an expression and "assigns" it to a destination; the expression may involve arithmetic computation, the result being assigned to an arithmetic-type destination, or string operations with the result assigned to a string destination; and provided the two structures are of matching type this definition of assignment also includes the possibility of copying a whole array or record as a single value.

The elementary constituents of expressions are primaries; suitable primaries can be combined using operators. When an expression involves several different operators, the order in which the operations are carried out is determined by the relative **precedence**, for instance multiplying operators have higher precedence than adding operators. You can use parentheses to modify the precedence ordering; a subexpression in parentheses is treated as another kind of **primary** or elementary item. This recursive form of definition allows you to build up expressions to any degree of complexity. For more details see operator precedence.

Most expressions are found within the statement-part of a program or procedure, that is, the part which specifies the actions. You can, however, construct expressions whose constituents are all constant, and such constant expressions can appear in other contexts. It may be convenient, for instance, to define the dimensions of an array, or an initial value, using operators, rather than as a single constant.

Primary operands

Expressions allow you to combine operands using operators. The simplest forms of operand are primaries, of which there are several kinds:

Variables (more correctly, variable-accesses)

Constants

Function results

Constructors

Subexpressions

Other primaries

Variables

The term variable-access is used to describe the process which starts with the name of a variable. If you wish to access the whole (or "entire") variable, its name is all that is needed. On the other hand, if the variable is an array, you may wish to access one element, or if it is a record to access a field. In formulating a variable-access, you can use a combination of indexing (`[index]`), field selection (`.fieldname`) and pointer following (`^`) to find the sub-item in which you are interested.

As an example, suppose you have created a record in the heap and placed a pointer to it in a variable called `prec`. One field of the record, called `avals`, is an array of integers. To access element `k` of this array, you write:

```
    prec^.avals[k]
```

You are describing the process of taking the pointer `prec`, following it to the record, getting to the field `avals`, and selecting element `k` of the array. Notice that this is an access of type integer; as a reference within an expression it produces an integer value, and it can also appear as a destination for storing an integer result.

If you were to leave out the index from the previous example, it would become a reference to the whole array, which is appropriate when assigning or copying as a structure. Similarly, the access `prec^` refers to the whole record.

An index value is another place where an expression can appear. Instead of `[k]` above, it would be possible to write say `[4*k-10]`, so long as this was indeed a valid index value. The type of an index expression must be compatible with the index in the definition of the array type.

Constants

The constants used within expressions may appear explicitly, for example `5`, `16#FFFF`, `2.75` or `'OK'` (see Numbers and Character strings). Alternatively, they can have been defined as named constants in a previous CONST definition, and appear in expressions by name. It is a good idea to employ named constants in any situation in which the same value or related values are needed in different places; named constants can be defined by constant expressions, so any related values can be written in forms that express the connections between them.

Constants can only be used in situations where their type is appropriate; you cannot for instance multiply or compare a day of the week and an integer constant. The value NIL is the only constant of pointer type, and is compatible with all defined pointer types.

Constant values of set or structured types can be written using the set-constructor, array-constructor and record-constructor notations with constant constituents. The notations are described in Set types, Array constructors and Record constructors. Again, it is often a good idea to define constructors as named constants.

Functions

When you call a Pascal function, it returns a value which can be used as a primary in an expression. In standard Pascal, functions can return numeric or pointer values; in Extended Pascal they can also return strings, arrays and records. When the result is numeric, it can take part in operations, just as a variable or constant can do. For example, the predefined function rand returns a random real value in the range 0.0 to 1.0, and the function trunc takes a real value and returns the integer part. The expression `trunc(rand*1000))` returns a random integer value in the range 0 to 999.

The results returned by string-type functions can take part in string-type expressions, for instance with the predefined function substr you could combine parts of two strings `s1` and `s2` like this:

```
     substr(s1,1,5) + substr(s2,6)
```

In Extended Pascal you can use the result of a function of pointer, array or record type as the starting point for a variable access, instead of starting with a variable name, but it is usually more efficient to call the function separately and store the result.

Constructors

You can use a constructor within an expression to introduce a value of a composite type, and in this context it can include constituents whose values are determined at run-time. One kind is the set constructor, introduced in the description of set types. Pascal set types allow you to work with groups of values of an ordinal type such as integer, char or an enumerated type, which are not consecutive. For example, in classic Pascal you can use a set as a parameter to procedure which reads and processes text, to tell it which characters may terminate the next operation. In Extended Pascal you can use a FOR statement which iterates through a set of values defined by a set, so you could for instance write a statement such as:

```
    FOR day IN [Monday,Wednesday,Friday] DO
     NoFreeLunch (day);
```

where `day` is a local variable of type DayOfWeek.

Array constructors and record constructors were introduced in the description of array and record types. There the examples showed constructors made up of constant values, but within expressions the individual elements can have run-time values (which are themselves, in fact, expressions). If you are for example passing a record as a value parameter, you can write it as a constructor instead of introducing a local variable. If the type `rect` has fields `top`, `bottom`, `left` and `right`, the following statement calls procedure `draw` with a parameter of type `rect`:

```
    draw (rect[top:m; bottom:m+height;
            left:n; right:n+width]);
```

Array and record constructors are not available in classic Pascal, but are part of the Extended Pascal standard.

Subexpressions

When you are building up an expression, typically an arithmetic expression, combining values by means of operators, the order in which the operations are carried out is decided by the relative precedence of the operators. When you need to override the normal precedence, you can do so by putting part of the expression (a subexpression) in parentheses. Because DIV, for example, has higher precedence than + or -, you write `(j+k)DIV(n-5)` to cause `j+k` and `n-5` to be evaluated before the DIV; without the parentheses, the division would be performed first. Pascal formalises this important provision by classifying a subexpression as a form of primary, like variable and constant.

Subexpressions are needed quite regularly in forming compound conditions. For one thing, as explained in the description of **precedence**, the relational operators have low position, and a condition such as `i<10` or `day=Friday` must be parenthesised if it is to be combined. But also, in the precedence order, AND is higher than OR, so that given the expressions:

```
      condition1 AND condition2 OR condition3
      condition1 AND (condition2 OR condition3)
```

the parentheses in the second case override the natural ordering in the first, which places AND before OR. When the individual conditions also require parentheses, a visual layout that shows the intention can be specially helpful.

It is incidentally quite permissible to use parentheses where they are not strictly needed, simply to illustrate your intention. You could for instance write:

```
    IF  ((s[1] = '*') AND (s[k] = lch1)) OR
      (s[k-1] = lch2)  THEN ...
```

to show that both the first two conditions must be true to make the combination true.

Other primaries

There are three further forms of primary that are needed only in particular situations. In the description of string types, the use of a VAR formal parameter of type **string** included a reference to the capacity of the actual parameter as "parametername.capacity". This reference is an example of a special form of primary technically known as schema-discriminant. If you have defined any schema type of your own, you can obtain the values of discriminants in the same way, quoting the name of the formal discriminant.

Another form of primary occurs in the conformant array parameter option that is available in both unextended and extended Pascal. The bound identifiers which appear in the parameter list associated with the conformant array can be referred to within the procedure or function as another form of primary. They allow the code within the procedure or function to adapt to the bounds of the index type of the actual parameter.

The last situation occurs in the definition of a schema type. When the new type includes for instance an array index which involves the value of a formal discriminant, the reference is made by quoting the name of the discriminant. In the following definition:

```
  TYPE  vec(last: integer) = ARRAY [1..last] OF real;
```

the reference to `last` in the array index is called a discriminant-identifier, which is a form of primary that can only be used in that particular context.

Operators

You use operators to modify or combine values within an expression. The majority are binary operators (so called because they take two operands, not because these are binary values). Familiar examples are arithmetic operators such as * (multiply), logical operators such as `OR`, and relational operators such as < (less than). The unary operators by contrast take a single value; there are for practical purposes two of these, – (negate) and `NOT`. The relative priorities of different operators are discussed in

Operator precedence

See the following sections for further details:

Arithmetic operations

Boolean (logical) operations

Relational (comparison) operations

String operations

Set operations

Arithmetic operations

Arithmetic operators allow you to compute integer, real and complex values. Integer arithmetic operations take integer-type values and produce an integer result; integer subrange values (such as byte or word, or subranges you have defined yourself) are automatically widened before the operation takes place. Real (floating-point) arithmetic operations take real values, or one real and one integer value, and produce a real-type result; if there is an integer operand, it is converted before the operation. Complex arithmetic operations take complex values, or one complex value and one real or integer value, and produce a complex result.

In many cases, the same symbol is used to specify an operation whichever arithmetic types are involved, and the mode of the actual operation is derived from the operands. You can therefore look at the statements in the previous paragraph another way, and say that there is a progression from integer to real and then to complex. If the operands have different types, the junior is promoted to the type of the senior, and the operation is carried out in the senior mode. Given an integer variable `ivar` and a real variable `rvar`, the statements

```
    ivar := 22;
    rvar := ivar + 5.25;
```

store 22 in `ivar`, then convert the integer value to floating-point, add 5.25, and store 27.25 in `rvar`. Some operators, however, have special rules (for instance, they apply in integer mode only), and these are shown in the next paragraph.

The symbols that you can use to specify arithmetic operations are:

+	add (integer, real or complex)
-	subtract or negate (integer, real or complex)
*	multiply (integer, real or complex)
/	real divide (see note 1)
`DIV`	integer divide (see note 1)
`MOD`	modulus (integer only)
**	exponentiation (see note 2)
`POW`	exponentiation (see note 2)
`REM`	remainder (integer only)
`SHL`	shift left (integer only, note 3)
`SHR`	shift right (integer only, note 3)
`AND`	bitwise AND (integer only)
`OR`	bitwise inclusive OR (integer only)
`XOR`	bitwise exclusive OR (integer only)

Notes. (1) The symbol / specifies a real-mode operation; if both operands are integer, both are converted before the division. The symbol `DIV` specifies integer division with truncation; both operands must be integer. (2) The symbol ** specifies real or complex mode exponentiation; the power may be integer or real. The symbol `POW` specifies integer, real or complex exponentiation to an integer power. (3) The shift operations take the left-hand operand and shift by the number of places (maximum 31) specified by the right-hand operand.

Integer values produced during computation must as a rule lie between minus maxint and plus maxint (maxint = 2147483647). If this range is exceeded, the result is not predictable; however, you can request the compiler to insert checks and signal if such overflows occur. There is an exception to the rule in this implementation: when an integer multiply is immediately followed by an integer divide, an extended product is generated, and only the result of the divide must be within range.

Floating-point arithmetic is performed at extended precision in Intel processors, and intermediate results will almost never overflow. It is possible for overflow to occur when a result is stored in a real (or more particularly a shortreal) variable, and an exception is raised if this occurs. Note that the built-in function **pi** produces an extremely precise value of the constant.

A number of the symbols used to specify arithmetic operations are also used with different meanings in conjunction with other types of operand (see String operations, Set operations and Boolean operations). The relative precedence of operators is described in Operator precedence.

The operators +, -, *, /, `DIV` and `MOD` are defined in standard Pascal; ** and `POW` are defined in the Extended Pascal standard. The others are local additions.

Boolean operations

Boolean (logical) operators allow you to combine Boolean values to express compound conditions or logical results. In this usage, the operand or operands are Boolean type, and the result is also Boolean.

The symbol `NOT` is a unary operator; `AND` and `OR` are binary operators. They are all defined in standard Pascal, and have the "obvious" meanings (see below). Alternative forms `AND_THEN` and `OR_ELSE` were introduced in Extended Pascal, which require that the remainder of the expression is not to be evaluated when the final value is already clear. You can write something like this:

```
    WHILE (p <> NIL) AND_THEN (p^.item < 100) DO ...
```

to ensure that if the pointer `p` is `NIL`, no attempt is made to use its value. (This implementation in fact treats plain `AND` and `OR` in the same way, but other implementations may not.)

The relative priorities of operators are described in Operator precedence, where it will be seen that `NOT` has a high priority. It follows that the two expressions:

```
        NOT bool1 OR bool2
        NOT (bool1 OR bool2)
```

are both perfectly legal but mean different things. In the first, the `NOT` is done first, in the other, the `OR` is evaluated first. Similar considerations arise from `AND` having higher precedence than `OR`; use parentheses to override the precedence ordering if necessary. (Good programmers have been known to confuse themselves in this area.)

The result of the operation `b1 AND b2` is true only if `b1` and `b2` are both true, and the result of `b1 OR b2` is true if either operand is true. A local operator XOR is provided that yields true if one or other, but not both, of its operands is true.

Relational operations

Relational operators are used to compare pairs of values, and (less often) to check whether a value is present in a set, or an object is of a certain type. The operands in a comparison can be of various types, but they must be compatible, for instance both numeric, or both character type. With some operand types, only a subset of relational operations make sense and are permitted.

When comparing numeric values, relational operators follow the same rule as arithmetic operations in converting integer to real and real to complex before the operation; complex values can only be compared equal or unequal. Character or string values are compared on the basis of the unsigned ordinal values of the characters. Comparisons of strings of unequal length are performed as if the shorter was extended to the length of the longer with space characters; see eq and related functions for string comparisons taking account of length. Unicode strings can only be compared using these functions.

The operators = (equal) and <> (not equal) can be used to compare two numeric values, characters, strings, or items of enumerated, pointer or set types. Operators < (less than) and > (greater than) can be used with integer or real values, characters, strings or enumerated types. Operators <= (less-or-equal) and >= (greater-or-equal) can be used with integer or real values, characters, strings, enumerated types and sets; when comparing sets, the test s1<=s2 is true if s1 is a subset of s2, and s1>=s2 is true if s2 is a subset of s1.

The operator `IN` takes an ordinal value and a set, and returns true if the value is a member of the set. The value and the base type of the set must be compatible. You can sometimes reduce multiple comparisons by using `IN`, for example:

```
      item IN [3..5, 7, 8, 10]
      ch IN ['A','E','I','O','U']
```

which are true if `item` has any of the values shown, or `ch` is one of the vowels.

The operator `IS` takes an object reference or exception record name, and a class-name or exception-name; for details see LRM section 13.8.6 or 14.3.5.

Relational operations have a low relative precedence, and parentheses are needed when they are combined using `AND`, `OR` and `NOT` (see Boolean operators). The comparison operators each have a complement, which usually allows you to avoid using `NOT`, but the same does not apply to `IN`, and you must write:

```
      NOT (item IN [3..5, 7, 8, 10])
```

to reverse the example above.

String operations

In Extended Pascal, the treatment of strings and individual characters employs the concept of string-type values. References to string or character variables, and string or character constants, all yield these string-type values. Many of the operations you perform on strings involve use of the string handling procedures and functions, which often take string-type values as parameters or return string-type results; however the symbol + is used for the operation of concatenating (joining) string-type values. You can obtain a substring from a string variable using the index notation **s[i1..i2]**, or from a string-type value using the substr function.

Any string expression returns a string-type value, which can be assigned to a string destination, written to a textfile, or passed as a value parameter. A string variable is declared with a particular **capacity**, for instance `string(50)` has capacity 50; it can hold any number of characters from 0 to its capacity, and the current **length** of the contents is maintained along with with the characters, and can be obtained by calling the length function. Assignment of a value to a string variable defines the length as well as the characters, so for example the two variables are declared as

```
    VAR   s1,s2: string(50);
```

each has a capacity of 50 characters. You can assign values to these variables like this:

```
       s1 := 'abcde';
       s2 := s1 + 'fgh';
```

after which `length(s1)` is 5, `length(s2)` is 8, and `s2[4..7]` is 'defg'.

You can also assign to a so-called fixed string, defined as a packed array of char of specified size, and in this case if the value does not occupy the whole array, any remaining positions are filled with spaces. To illustrate this, if you declare a variable as:

```
    VAR   buffer: PACKED ARRAY [1..100] OF char;
```

then the statement `buffer:=' ';` will fill the whole array with spaces. A substring defined with the notation **s[i1..i2]** can be used as a destination as well as a reference; if a shorter value is assigned, the remainder of the substring is space-filled, as with a fixed string.

The shortstring type, and several of the string-handling routines, are local additions, included to facilitate the transfer of programs from other implementations. Variables declared as shortstrings can be used in much the same way as the default strings described above, though they should be avoided in new programs. The wchar and widestring types, on the other hand, are provided particularly for this implementation to enable you to accommodate and manipulate Unicode. A number of the string handling routines accept widestring variables, andyou can assign them and pass them as parameters, but the + operator is not available. For further details see the description of widestring type.

Set operations

Pascal allows you to define set types; you can declare variables of such types, and specify the operations of union, difference and intersection which are described below. You can assign sets, and pass them as parameters, just as with other types. Most programming languages do not provide facilities for working with sets explicitly, but you may nevertheless find the ideas behind them familiar.

To illustrate some of the possibilities, assume the definition

```
    TYPE  charset = SET OF char;
```

and the declarations

```
    VAR   ucase: charset VALUE ['A'..'Z'];
          lcase: charset VALUE ['a'..'z'];
          alpha,alphanum: charset;
```

To combine the members of two sets, you can use the **union** operation, denoted by +

```
    alpha := ucase + lcase;
    alphanum := alpha + ['0'..'9'];
```

The union operation forms a set value containing the members that are present in either (or both) of the operands; the **difference** operation `s1-s2` forms a set containing all the members in s1 that are not also in s2 (that is, it "takes away" s2 from s1). The **intersection** operation `s1*s2` produces the members that are present in both s1 and s2.

In this implementation, `'A'..'Z'` implies all the upper case letters in the Latin alphabet. As an alternative, consider the following method, where `ch` is a character variable.

```
    Alpha := [ ];
    FOR ch := ' ' TO maxchar DO
     IF IsAlpha(ch) THEN alpha := alpha+[ch];
```

This shows how you can use the empty set [ ], and how a constructor can include a variable component such as `[ch]`. In fact, a constructor can bring together both constant and variable elements. The function IsAlpha is a local extension which returns true if given an alphabetic character, including non-Latin letters. It also adapts to the character code in use, and may even produce a different set depending upon where you run the program.

See relational operations for a description of the `IN` operator that allows you to test whether a member is present in a set.

Operator precedence

The relative precedence of operators within an expression or subexpression determines the order in which the operations will be carried out. There are five categories, which in most situations produce the least surprising outcome. For example, in the expression

```
    a * 5 + b * 20
```

the two multiplications `a*5` and `b*20` have higher precedence that the addition, so both are performed, and the two resulting values then added together. If you want addition to precede multiplication, you can use parentheses like this:

```
    (a + 5) * (b + 20)
```

This requires `a+5` and `b+20` to be calculated and the two resulting values to be multiplied. Sometimes, this effect is described by saying that * has greater "binding strength" than +.

Starting with the highest (most tightly binding), the precedence categories are as follows.
  NOT
  Exponentiating operators (** and POW) and IS
  Multiplying operators (*, /, DIV, MOD, REM, SHL, SHR, AND, AND_THEN)
  Adding operators (+, -, ><, OR, OR_ELSE, XOR)
  Relational operators (=, <>, <, <=, >, >=, IN)
Within each group, a series of operations is generally carried out left-to-right, though this is not required by the Pascal standards. For example, in `j+k-n` the addition of j and k is done first, and n is subtracted from the result. The order seldom matters in practice, but just occasionally, in integer working, an overflow may be avoided by rearranging the sequence.

In most cases these groupings are similar to those found in other languages such as Fortran, and lead to the most convenient forms of expression. One point to be aware of, however, is that relational operators have lower precedence than AND and OR, and when writing compound conditions parentheses are regularly needed, for instance:

```
    IF (a < 5) AND (b > 20) THEN ...
```

Without the parentheses, `5 AND b` would be the first operation to be attempted.

Constant expressions

An expression in which all the ingredients are constant, either explicit literals such as 57 or 'Monday', or names previously defined as constants, is a constant expression. Such expressions are evaluated at compile time. For one thing, this saves on program size and improves execution speed, but perhaps more importantly, constant expressions can be used in almost all the situations where earlier versions of Pascal required a single constant. You can for instance define a size and related index range such as these

```
    CONST buffsize = 1000;
    VAR   buffer: ARRAY [0..buffsize-1] OF char;
```

and some other uses are mentioned below.

Constant expressions are included in the Extended Pascal standard, and are also found in several other Pascal implementations. As defined in the standard, the facilities are very comprehensive; a constant expression can include any operator and almost all built-in functions (such as `length`, `abs` and `max`) provided any arguments are constants, though not functions which provide run-time information such as `eof` or `addrof`.

One particular application of constant expressions is in building up string constants, as

Built-in Procedures and Functions

String handling

File handling

Mathematical and numeric

Memory allocation

Date and time

Machine level operations

Character operations

Miscellaneous

String handling

These procedures and functions allow you to perform operations on conventional strings and in some cases on Unicode strings also. A number of them are included to provide continuity with older Pascal implementations.

concat   copy   delete   eq   ge   gt   index   insert

le   length   lt   ltrim   ne   pos   readstr   setlength

substr   trim   WideToStr   writestr

File handling

These procedures and functions allow you to perform operations on sequential and direct-access files. There are operations for associating a Pascal file variable and an external file, preparing it for input or output operations, and then for reading or writing.

bind   binding   close   connect   echo   empty

eof   eoln   extend   get   HandleOf   LastPosition

OpenRead   OpenWrite   page   position   put

read   readln   reset   rewrite

SeekRead   SeekUpdate   SeekWrite

unbind   update   write   writeln

Mathematical and numeric

These procedures and functions (most are functions) provide mathematical and other numeric operations.

abs arccos arcsin arctan arg cmplx cos cosh

dec exp frac im inc int ln max min

odd ord pi polar rand re round seed

sin sinh sqr sqrt succ tan tanh trunc

Memory allocation

These procedures and functions are used to request space and return it, and to make related enquiries.

dispmem dispose HeapUsed

memavail new newmem

Date and time

In this group are the built-in routines relating to date and time. A number of other operations are available in the library.

date GetTimeStamp time

Machine level operations

This group contains routines for working at the machine level when the situation demands it; a number of them are needed for API programming in particular. In earlier Pascal implementations, similar facilities were used for crossing type boundaries, but this implementation provides clearer ways of doing this in new programs.

addrof   clearbit   FieldOffset   flipbit

move   setbit   sizeof   taddrof   testbit

Character operations

These are operations on individual characters; the LowerCase and UpperCase functions are also available for character strings.

chr   IsAlpha   IsDigit

LowerCase   UpperCase   wchr

Miscellaneous

This group contains a variety of routines which do not fall into a specific category.

assert   card   exit   halt   InfoBox   pack   raise   RaiseUser

return   StackUsed   stkavail   unpack   YesNoBox

abs

Kind: Built-in function (all levels)

Group: Mathematical/Numeric

Format: `abs(x)`

Argument x can be a value of any arithmetic type. Function abs(x) returns the absolute value of x. If x is integer or real, result is same type; if x is complex, result is real. (LRM 9.3.1)

addrof

Kind: Built-in function (local)

Group: Machine level

Format: `addrof(access)`

Parameter access is a variable-access or string constant; function `addrof` returns a result of type ptr representing the address of the access. The use with string constants is intended in particular for API calls, and a terminating null is added automatically. (LRM 9.7.4)
See also taddrof.

arccos

Kind: Built-in function (local)

Group: Mathematical/Numeric

Format: `arccos (x)`

Argument integer or real (-1 <= x <= 1).
Result real (radians, range 0 to pi).

arcsin

Kind: Built-in function (local)

Group: Mathematical/Numeric

Format: `arcsin (x)`

Argument integer or real (-1 <= x <= 1).
Result is real (radians, range -pi/2 to +pi/2).

arctan

Kind: Built-in function (all levels)

Group: Mathematical/Numeric

Format: `arctan (x)`

Argument x is integer or real expression.
Result is real (radians).

arg

Kind: Built-in function (EP standard)

Group: Mathematical/Numeric

Format: `arg (z)`

Argument of complex expression z; result real (radians).

assert

Kind: Built-in procedure (local)

Group: Miscellaneous

Format: `assert (b [,n] [,s])`

Optionally compiled assertion. Argument b is Boolean expression, optional arguments `n` and `s` are integer and string values. If compiled and b is false, an exception is raised. (LRM 9.7.9)

bind

Kind: Built-in procedure (EP standard)

Group: File handling

Format: `bind (f, bt)`

Bind `f` to the external entity defined by `bt`. Normally, f is a bindable file; bt is BindingType. (LRM 9.2.3.3)

See also: OpenRead OpenWrite

binding

Kind: Built-in function (EP standard)

Group: File handling

Format: `binding (f)`

Return status of f. Normally, f is a bindable file; result is always BindingType. (LRM 9.2.3.2)

card

Kind: Built-in function (EP standard)

Group: Miscellaneous

Format: `card (t)`

Returns the cardinality of set expression t (that is, the number of members present in t); result is integer.

chr

Kind: Built-in function (all levels)

Group: Character operations

Format:
```
chr(i)
chr(wch)
```

Function `chr` returns the 8-bit character corresponding to integer argument i (i >= 0) or Unicode character wch. The result is char type. (LRM 9.3.7)

See also: wchr

clearbit

Kind: Built-in function (local)

Group: Machine level

Format: `clearbit (loc,n)`

Clear bit number n at location loc to 1 and return the previous value as a Boolean result (0=>false, 1=>true). See also flipbit, setbit, testbit. (LRM 9.6.1)

close

Kind: Built-in procedure (local)

Group: File handling

Format: `close (f)`

Close file f (any file type). Files are normally closed automatically, `close` is needed only in special situations, see LRM 9.2.8.1.

cmplx

Kind: Built-in function (EP standard)

Group: Mathematical/Numeric

Format: `cmplx(x,y)`

Return the complex value having real and imaginary parts x and y (x, y real or integer). Result is complex type. Used with constant x and y to define complex constants.

See also: re im polar

concat

Kind: Built-in function (local)

Group: String handling

Format: `concat(s1, s2 [,s3 ...])`

Return the string obtained by concatenating the string values s1, s2 ... Result is general string type.
Included for compatibility with older implementations; equivalent to + operator in EP.

connect

Kind: Built-in procedure (local)

Group: File handling

Format: `connect(f, h)`

Connect file f to standard handle h. Parameter f is any file type (normally text), h takes values 0 (input), 1 (output) or 2 (error). Retained for compatibility with older implementations. Valid in console mode programs only.

copy

Kind: Built-in function (local)

Group: String handling

Format:
```
copy(s,inx,num)
copy(objref)
```

The first form is retained for continuity with older Pascal implementations; it returns a string value obtained by taking num characters from string s, starting at position inx. In EP the equivalent is substr(s,inx,num).

The second form makes a copy of an object. Parameter `objref` is a reference to an object. A new object of the same object type is created and the fields of the original object are copied to it; a reference to the new object is returned.

cos

Kind: Built-in function (all levels)

Group: Mathematical/Numeric

Format: `cos(x)`

Returns cosine of x (abs(x) < 2.14E9). Argument x is any arithmetic type; if x is integer or real, result is real, if x is complex, result is complex. (LRM 9.3.2)

cosh

Kind: Built-in function (local)

Group: Mathematical/Numeric

Format: `cosh(x)`

Returns the hyperbolic cosine of x (where abs(x) < 710). Argument x is integer or real, result is real. (LRM 9.3.3)

date

Kind: Built-in function (EP standard)

Group: Date and time

Format: `date(ts)`

Parameter `ts` is a TimeStamp. The function returns the date defined by ts as a string value. (LRM 9.5.3)

The contents of ts may have been obtained by GetTimeStamp, or may alternatively have been returned by a library routine or set individually by the program. Field DateValid must be true.

dec

Kind: Built-in procedure (local)

Group: Mathematical/Numeric

Format: `dec(v [,n])`

Parameter v is a variable-access of ordinal type (integer, char, enumerated etc). It is decremented by 1 or by the value of the optional integer n.

See also: inc

delete

Kind: Built-in procedure (local)

Group: String handling

Format: `delete(s, inx, n)`

Delete n characters from string variable `s`, starting at character `inx`. Can also be applied to short or wide strings.

See also: insert

dispmem

Kind: Built-in procedure (local)

Group: Memory allocation

Format: `dispmem(p, size)`

Dispose heap space of size bytes at position p^, obtained by procedure newmem. Included for compatibility, alternative methods are recommended in new programs. (LRM 9.4.5)

dispose

Kind: Built-in procedure (all levels)

Group: Memory allocation

Format:
```
dispose(p)
dispose(p, tag1 [,tag2 ...])
```

Dispose heap space at position p^, obtained by procedure new. The version for variant records quoting tag values must match the corresponding new. (LRM 9.4.1-4)

echo

Kind: Built-in procedure (local)

Group: File handling

Format: `echo(txf, onoff)`

This procedure is for use in console mode programs. Parameter `onoff` is Boolean. When true, output to textfile txf is echoed to the standard error handle. (LRM 9.2.9.1)

empty

Kind: Built-in function (EP standard)

Group: File handling

Format: `empty(ntf)`

Parameter `ntf` is a direct-access (indexed) file, already associated with an external file. This Boolean function returns true if the associated external file is empty.

eof

Kind: Built-in function (all levels)

Group: File handling

Format: `eof [(f)]`

Boolean function `eof` returns true if the external file associated with parameter f is at end-of-file. If no file is specified, standard input is implied. (LRM 9.2.4.1)

See also: eoln

eoln

Kind: Built-in function (all levels)

Group: File handling

Format: `eoln [(txf)]`

Boolean function `eoln` returns true if the external textfile associated with parameter txf is at an end-of-line marker. If no file is specified, standard input is implied. (LRM 9.2.4.1)

It is an error to apply this function to a file which is at end-of-file. Check eof first.

eq

Kind: Built-in function (EP standard)

Group: String handling

Format: `eq(s1,s2)`

This function compares two string values `s1` and `s2` taking account of lengths (see LRM 9.1.5) and returns a Boolean result which is true if s1 and s2 are equal. It can also be used to compare widestring variables.

exit

Kind: Built-in procedure (local)

Group: Miscellaneous

Format: `exit`

Causes immediate return from the current procedure or function; in the case of a function, the most recently assigned value of the function variable is returned. (LRM 9.7.6) If used within the statement part of a main program, it causes a jump to the end of the statement part. See also halt and return.

exp

Kind: Built-in function (all levels)

Group: Mathematical/Numeric

Format: `exp(x)`

Returns the exponential of x (x < 710). Argument x is any arithmetic type; if x is integer or real, the result is real, if x is complex, the result is complex.

extend

Kind: Built-in procedure (EP standard)

Group: File handling

Format: `extend(f)`

Prepares file f for writing additional data following the current contents. (LRM 9.2.3.8)

See also: rewrite

FieldOffset

Kind: Built-in function (local)

Group: Machine level

Format: `FieldOffset(t,fld)`

This function takes as arguments a record typename `t` and a fieldname `fld` in the record; it returns the offset of the field within the record. It is included primarily for use in API calls, where the offset value is occasionally needed. (LRM 9.7.5)

flipbit

Kind: Built-in function (local)

Group: Machine level

Format: `flipbit(loc, n)`

Reverse bit number `n` at location `loc` to 1 and return the previous value as a Boolean result (0=>false, 1=>true). See also clearbit, setbit, testbit. (LRM 9.6.1)

frac

Kind: Built-in function (local)

Group: Mathematical/Numeric

Format: `frac(x)`

Returns fractional part of real argument x; result is real. See also int. (LRM 9.3.13)

ge

Kind: Built-in function (EP standard)

Group: String handling

Format: `ge(s1, s2)`

This function compares two string values s1 and s2 taking account of lengths (see LRM 9.1.5) and returns a Boolean result. It can also be used to compare widestring variables.

get

Kind: Built-in procedure (all levels)

Group: File handling

Format: `get(f)`

This is the basic operation to advance a file which is in inspection (input) or update mode to the next element, making it available in the file buffer. (LRM 9.2.4.2)

See also: put read

GetTimeStamp

Kind: Built-in procedure (EP standard)

Group: Date and time

Format: `GetTimeStamp(ts)`

Parameter `ts` is of type TimeStamp. The current date and time ("local time") is returned. (LRM 9.5.2)

gt

Kind: Built-in function (EP standard)

Group: String handling

Format: `gt (s1, s2)`

This function compares two string values s1 and s2 taking account of lengths (see LRM 9.1.5) and returns a Boolean result. It can also be used with widestring variables.

halt

Kind: Built-in procedure (EP standard)

Group: Miscellaneous

Format: `halt [(exitcode)]`

Terminate program execution without performing module finalization. The non-standard parameter exitcode defines the value that will be returned from the process. (LRM 9.7.3) If used within the main program, the non-standard procedures exit and return also cause termination, with different effects.

HandleOf

Kind: Built-in function (local)

Group: File handling

Format: `HandleOf(f)`

Returns the "handle" of an open file, or -1. (LRM 9.2.9.2)

HeapUsed

Kind: Built-in function (local)

Group: Memory allocation

Format: `HeapUsed`

Returns an estimate of heap occupancy (bytes). (LRM 9.4.6)

See also memavail.

im

Kind: Built-in function (EP standard)

Group: Mathematical/Numeric

Format: `im(z)`

Returns the imaginary part of complex argument z.

See also: cmplx re

inc

Kind: Built-in procedure (local)

Group: Mathematical/Numeric

Format: `inc(v [,n])`

Parameter `v` is a variable-access of ordinal type (integer, char, enumerated etc). It is incremented by 1 or by the value of the optional integer n.

See also dec.

index

Kind: Built-in function (EP standard)

Group: String handling

Format: `index(s1,s2)`

Search for the first occurrence of string `s2` in `s1` and return the index number, or zero if not found. Parameters s1 and s2 are general string values. (LRM 9.1.2)

If an occurence has been found, the remainder of s1 can be examined using for instance `substr(s1,first+length(s2))` in place of s1.

See also: pos substr

insert

Kind: Built-in procedure (local)

Group: String handling

Format: `insert(sval, svar, index)`

Insert string value `sval` into string variable `svar` at position `index`, moving up any

IsAlpha

Kind: Built-in function (local)

Group: Character operations

Format: `IsAlpha(ch)`

Parameter `ch` may be of type char or wchar. Returns a Boolean result, which is true if `ch` is an upper-case or lower-case letter. (LRM 9.7.7)

IsDigit

Kind: Built-in function (local)

Group: Character operations

Format: `IsDigit(ch)`

Parameter `ch` may be of type char or wchar. Returns a Boolean result, which is true if

LastPosition

Kind: Built-in function (EP standard)

Group: File handling

Format: `LastPosition(ntf)`

Parameter `ntf` is a direct-access (indexed) file, already associated with an external file. This function identifies the last element of the external file. The result type is the type of the index. (LRM 9.2.6.4)

le

Kind: Built-in function (EP standard)

Group: String handling

Format: `le(s1, s2)`

This function compares two string values s1 and s2 taking account of lengths (see LRM 9.1.5), and returns a Boolean result which is true if s1 is less than or the same as s2. It can also be used to compare widestring variables.

length

Kind: Built-in function (EP standard)

Group: String handling

Format: `length(s)`

Returns the current length of string `s`. Parameter s may be a conventional string variable or expression, or a widestring variable. Result is integer. (LRM 9.1.1)

ln

Kind: Built-in function (all levels)

Group: Mathematical/Numeric

Format: `ln(x)`

Returns natural logarithm of x (x > 0). Argument x is any arithmetic type; if x is integer or real, result is real, if x is complex, result is complex.

LowerCase

Kind: Built-in function (local)

Group: Character operations

Format: `LowerCase(ch)`
`LowerCase(str)`

Parameter `ch` may be a character or wide character, parameter `str` is a string expression; the result is same type as the parameter. An upper-case letter is changed to the lower-case equivalent, any other character being returned unchanged. A string is returned with any upper-case letters changed to lower-case equivalents. See also UpperCase. (LRM 9.7.8)

lt

Kind: Built-in function (EP standard)

Group: String handling

Format: `lt(s1,s2)`

This function compares two string values s1 and s2 taking account of length (see LRM 9.1.5) and returns a Boolean result which is true if s1 is less than s2. It can also be used with widestring variables.

ltrim

Kind: Built-in function (local)

Group: String handling

Format: `ltrim(sval)`

Returns a string-type result obtained from `sval` by removing any leading spaces. See also trim. (LRM 9.1.4)

max

Kind: Built-in function (local)

Group: Mathematical/Numeric

Format: `max(x,y)`

Arguments `x` and `y` can be of real type or any ordinal type (including for instance char or enumerated); their types must be compatible. The greater of x and y is returned. See also min. (LRM 9.3.11)

memavail

Kind: Built-in function (local)

Group: Memory allocation

Format: `memavail`

In earlier Pascal implementations this function returns an estimate of available heap space (bytes). It is retained only for compatibility, and is of limited use in the Win32 environment. See HeapUsed. (LRM 9.4.6)

min

Kind: Built-in function (local)

Group: Mathematical/Numeric

Format: `min(x,y)`

Arguments x and y can be of real type or any ordinal type (including for example char or enumerated); their types must be compatible. The smaller of x and y is returned. See also max. (LRM 9.3.11)

move

Kind: Built-in procedure (local)

Group: Machine level

Format: `move(src, dst, len)`

Copy `len` bytes from `src` to `dst`. This procedure is retained for compatibility with older implementations; in new programs almost all uses can be more safely achieved with Extended Pascal features. (LRM 9.6.2)

ne

Kind: Built-in function (EP standard)

Group: String handling

Format: `ne(s1,s2)`

This function compares two string values s1 and s2 taking account of length (see LRM 9.1.5) and returns a Boolean result, which is true if s1 and s2 are different. It can also be used with widestring variables.

new

Kind: Built-in procedure (all levels)

Group: Memory allocation

Format:
```
new(p)
new(p,tag1[,tag2 ...])
new(p,d1[,d2 ...])
```

Obtain heap space and define pointer p. When the domain type of p is a record with variants, tag values `tag1` etc can be supplied to obtain space for a specific variant. When the domain type is a schema name (including `string` or `widestring`), a discriminant value or values `d1` etc must be supplied and will determine the actual type and size.
See also dispose. (LRM 9.4.1-4)

newmem

Kind: Built-in procedure (local)

Group: Memory allocation

Format: `newmem(p,size)`

Obtain heap space of "size" bytes and define pointer p. Included for compatibility, alternative methods are recommended in new programs. See also dispmem. (LRM 9.4.5)

odd

Kind: Built-in function (all levels)

Group: Mathematical/Numeric

Format: `odd(i)`

Argument `i` is integer type; the function returns a Boolean result which is true if i is odd. (It avoids any possible implementation dependency.)

OpenRead

Kind: Built-in function (local)

Group: File handling

Format: `OpenRead(f,fname)`

Parameter `f` is a bindable file (of any component type), parameter `fname` is a string. OpenRead attempts to bind f to an external file fname and prepare it for reading (see procedure reset). It fails if the file fname does not exist. A Boolean result is returned, which is true if the operation was successful. See also OpenWrite below.

A file with read only attribute can be opened, but cannot later be extended or overwritten (see LRM 9.2.3.4).

OpenWrite

Kind: Built-in function (local)

Group: File handling

Format: `OpenWrite(f,fname)`

Parameter `f` is a bindable file (of any component type), parameter `fname` is a string. OpenWrite attempts to bind f to an external file fname and prepare it for writing (see procedure rewrite). If fname is valid but no such file exists, it is created; if fname is invalid (for example naming a nonexistant path) the function fails. A Boolean result is returned which is true if the operation was successful. See also OpenRead. (LRM 9.2.3.4)

ord

Kind: Built-in function (all levels)

Group: Mathematical/Numeric

Format: `ord(v)`

Argument `v` can be of any ordinal type (such as enumerated, char or wchar). Function ord returns an integer having the same ordinal value. (LRM 9.3.7)

See also: pred succ

pack

Kind: Built-in procedure (all levels)

Group: Miscellaneous

Format: `pack(unp,i,pkd)`

Move successive elements from an array of unpacked elements to an array of equivalent PACKED elements, starting at index i. (See LRM 9.7.1 for details.)

page

Kind: Built-in procedure (all levels)

Group: File handling

Format: `page [(txf)]`

Cause a new page to be taken on textfile txf. The file must be in generation (output) mode. If the parameter is omitted, standard file output is substituted. (LRM 9.2.5.5)

pi

Kind: Built-in function (local)

Group: Mathematical/Numeric

Format: `pi`

This function introduces a very precise value of the constant pi.

polar

Kind: Built-in function (EP standard)

Group: Mathematical/Numeric

Format: `polar(r,t)`

Return the complex value having `r` and `t` as magnitude and argument. See also cmplx. (LRM 9.3.6)

pos

Kind: Built-in function (local)

Group: String handling

Format: `pos(s1,s2)`

Parameters `s1` and `s2` are general string values. Function pos searches for the first occurrence of string s1 in s2, and returns the index number, or zero if no match is found. It is retained for compatibility; `pos(s1,s2)` is equivalent to the standard function `index(s2,s1)`. See index. (LRM 9.1.2)

position

Kind: Built-in function (EP standard)

Group: File handling

Format: `position(ntf)`

Parameter `ntf` is a direct-access (indexed) file. This function returns the index of the current element (ie `ntf^`) of the associated external file. The result type is the type of the index. See also LastPosition. (LRM 9.2.6.4)

pred

Kind: Built-in function (all levels)

Group: Mathematical/Numeric

Format: `pred(v [,n])`

Parameter `v` is of any ordinal type, including enumerated, char or wchar. The function returns a value having the type of v and which is one less than v, or n less than v if n is specified. The optional second parameter is not available in unextended Pascal. See also succ. (LRM 9.3.8)

put

Kind: Built-in procedure (all levels)

Group: File handling

Format: `put(f)`

This is the basic operation to advance a file in generation (output) or update mode to the next element, after writing the current contents of the buffer variable to the external file. (LRM 9.2.4.2)

See also: get write

raise

Kind: Built-in procedure (local)

Group: Miscellaneous

Format:
```
raise(ExcepName [,qualifier]
         [,intval] [,stringval])
```

Raise an exception (for instance to check exception handling code). (LRM 9.7.10) See also RaiseUser below.

RaiseUser

Kind: Built-in procedure (local)

Group: Miscellaneous

Format: `RaiseUser(intval [,stringval])`

Raise a User class exception. Parameters intval and stringval are reproduced in the notification, or can be examined in a TRY statement. (LRM 9.7.10)

rand

Kind: Built-in function (local)

Group: Mathematical/Numeric

Format: `rand`

Returns a pseudo-random real value; values are uniformly distributed over the range 0.0 to 1.0. See also seed. (LRM 9.3.12)

re

Kind: Built-in function (EP standard)

Group: Mathematical/Numeric

Format: `re(z)`

Returns the real part of complex argument z. See also cmplx, im.

read

Kind: Built-in procedure (all levels)

Group: File handling

Format: `read([txf,] v1 [,v2 ...])`
`read(ntf, v1 [,v2 ...])`

(a) Read one or more values from textfile txf and assign to parameters v1, v2 ... If txf is omitted, standard input is substituted. The file must be in inspection (input) mode. Conversion from external representation is performed automatically - see Textfile input, readln, readstr.

(b) Read one or more elements from non-text file ntf and assign to parameters v1, v2 ... The file must be in inspection (input) mode. Parameters must be assignment compatible with file element type. (LRM 9.2.5.1)

readln

Kind: Built-in procedure (all levels)

Group: File handling

Format: `readln[(txf)]`
`readln ([txf,] v1 [,v2 ...])`

Procedure readln advances an input textfile to the beginning of the next line; if no file is specified, standard input is substituted. There may be parameters, which obey the same rules as in procedure read, and are processed before advancing to the next line; that is, readln(v1,v2) is equivalent to read(v1,v2) followed by readln. (LRM 9.2.5.2)
See also Textfile input.

readstr

Kind: Built-in procedure (EP standard)

Group: String handling

Format: `readstr(str, v1 [,v2 ...])`

The string value `str` is treated as if it were a line of input from a textfile. The contents are transferred to parameters v1, v2 ... with appropriate conversions, as when reading from a file. See Textfile input. (LRM 9.1.6)

reset

Kind: Built-in procedure (all levels)

Group: File handling

Format: `reset(f)`

The file variable f must already be associated with an external file. It is put into inspection mode, that is, prepared for input operations. If the external file is empty, `eof(f)` becomes true, otherwise the buffer variable `f^` is positioned to the first file element. See also rewrite. (LRM 9.2.3.7)

return

Kind: Built-in procedure (local)

Group: Miscellaneous

Format: `return [(returnval)]`

Cause an immediate return from the current procedure or function. If it is an ordinal or pointer function, the return value can be supplied as an optional parameter. The return procedure can also be used in the statement part of a main program to immediately perform any module finalization and terminate execution. See also exit, halt. (LRM 9.7.6)

rewrite

Kind: Built-in procedure (all levels)

Group: File handling

Format: `rewrite(f)`

round

Kind: Built-in function (all levels)

Group: Mathematical/Numeric

Format: `round(x)`

Function round takes a real argument x and returns an integer which is the integral part of x after rounding away from zero. See also trunc. (LRM 9.3.4)

seed

Kind: Built-in procedure (local)

Group: Mathematical/Numeric

Format: `seed(seedval)`

The random number generator rand by default produces a different set of values at each execution of a program. Procedure seed initialises the generator, allowing the same sequence to be produced each time. Parameter seedval is an integer value. (LRM 9.3.12)

SeekRead

Kind: Built-in procedure (EP standard)

Group: File handling

Format: `SeekRead(ntf,elindex)`

Parameter `ntf` is a direct-access (indexed) file. SeekRead positions the file at element elindex and puts it into inspection mode in preparation for reading. See also SeekUpdate, SeekWrite. (LRM 9.2.6.1)

SeekUpdate

Kind: Built-in procedure (EP standard)

Group: File handling

Format: `SeekUpdate(ntf,elindex)`

Parameter `ntf` is a direct-access (indexed) file. SeekUpdate positions the file at element elindex and puts it into update mode in preparation for reading or writing. See also SeekRead, SeekWrite, update. (LRM 9.2.6.1)

SeekWrite

Kind: Built-in procedure (EP standard)

Group: File handling

Format: `SeekWrite(ntf,elindex)`

Parameter `ntf` is a direct-access (indexed) file. SeekWrite positions the file at element elindex and puts it into generation mode in preparation for writing. See also SeekRead, SeekUpdate. (LRM 9.2.6.1)

setbit

Kind: Built-in function (local)

Group: Machine level

Format: `setbit(loc, n)`

Set bit number `n` at location `loc` to 1 and return the previous value as a Boolean result (0=>false, 1=>true). Bits are numbered from zero, being the least significant. See also clearbit, flipbit, testbit. (LRM 9.6.1)

setlength

Kind: Built-in procedure (local)

Group: String handling

Format: `setlength(svar [,n])`

Normally, string operations keep track of the length of the current contents of a string variable automatically. Procedure setlength is provided for the unusual situations when the length must be set or overridden by the program, in particular when a string is returned by an API call, and avoids assumptions about the layout of strings in memory. Parameter `svar` is a string variable, which may be a conventional string, a shortstring or a widestring. Optional parameter `n` is the length to be established; if not present, svar is scanned for a null character. (LRM 9.1.8)

sin

Kind: Built-in function (all levels)

Group: Mathematical/Numeric

Format: `sin(x)`

Returns sine of x (abs(x) < 2.14E9). Argument x is any arithmetic type; if x is integer or real, result is real, if x is complex, result is complex. (LRM 9.3.2)

sinh

Kind: Built-in function (local)

Group: Mathematical/Numeric

Format: `sinh(x)`

Returns the hyperbolic sine of x (abs(x) < 710). Argument x can be integer or real, result is real. (LRM 9.3.3)

sizeof

Kind: Built-in function (local)

Group: Machine level

Format:
```
sizeof(t)
sizeof(v)
sizeof(t, tag1[,tag2 ...])
```

Function sizeof has a parameter which may be a type `t` or a variable `v`. It returns the default memory occupancy. The size must be one that is known at compile time. When the type is a record type with variants, tag values may be given as for procedure new. Note that for purposes of alignment or efficiency, more storage may be allocated than the minimum size for a type. (LRM 9.7.5)

sqr

Kind: Built-in function (all levels)

Group: Mathematical/Numeric

Format: `sqr(x)`

Argument x can be a value of any arithmetic type. Function sqr(x) returns the square of x (that is, x*x). The result type is the type of x. (LRM 9.3.1)

sqrt

Kind: Built-in function (all levels)

Group: Mathematical/Numeric

Format: `sqrt(x)`

Argument x can be a value of any arithmetic type. Function sqrt returns the square root of x. If x is integer or real, the result is real; if x is complex the result is complex. (LRM 9.3.2)

StackUsed

Kind: Built-in function (local)

Group: Miscellaneous

Format: `StackUsed`

Function StackUsed returns an integer value which is the number of bytes occupied by the stack. (If there is more than one thread active, each has its own stack, and the result relates to the stack of the current thread.) See also stkavail. (LRM 9.7.11)

stkavail

Kind: Built-in function (local)

Group: Miscellaneous

Format: `stkavail`

This function is retained for compatibility with older environments which provided a limited amount of space for the stack. It returns an integer result representing the number of bytes remaining for expansion. In a virtual memory environment, the amount of space available for the stack is very large, and this function is not usually helpful. See instead StackUsed. (LRM 9.7.11)

substr

Kind: Built-in function (EP standard)

Group: String handling

Format: `substr(sval,inx [,len])`

This function returns a substring extracted from the string value sval; the substring starts at position inx in sval. If the optional parameter len is present it defines the length of the substring, otherwise the substring contains the tail of sval starting at inx. (LRM 9.1.3)

succ

Kind: Built-in function (all levels)

Group: Mathematical/Numeric

Format: `succ(v [,n])`

Parameter `v` is of any ordinal type, including enumerated, char or wchar. The function returns a value having the type of v and which is one greater than v, or n greater than v if n is specified. See also pred. (LRM 9.3.8)

The optional second parameter is not available in unextended Pascal. In Extended Pascal, function succ can be used as inverse of ord for enumerated types; quote the first enumeration constant as v and the ordinal value as n.

taddrof

Kind: Built-in function (local)

Group: Machine level

Format: `taddrof(vaccess)`

Parameter vaccess is a variable-access; function taddrof returns a result of type ptr representing the address of the access. The access is "threatened" by the reference. It is intended in particular for API calls which require the address of a buffer to receive a result such as a string. (LRM 9.7.4)
See also addrof.

tan

Kind: Built-in function (local)

Group: Mathematical/Numeric

Format: `tan(x)`

Argument x can be integer or real ($x < 2.14E9$). Function tan returns the tangent of x, treated as radians. (LRM 9.3.3)

tanh

Kind: Built-in function (local)

Group: Mathematical/Numeric

Format: `tanh(x)`

Function tanh returns the hyperbolic tangent of integer or real argument x ($x < 710$). (LRM 9.3.3)

testbit

Kind: Built-in function (local)

Group: Machine level

Format: `testbit(loc, n)`

Test bit number n at location loc and return the value as a Boolean result (0=>false, 1=>true). Bits are numbered from zero, which is the least significant. See also clearbit, flipbit, setbit. (LRM 9.6.1)

time

Kind: Built-in function (EP standard)

Group: Date and time

Format: `time(ts)`

Parameter `ts` is a TimeStamp. The function returns the time defined by ts as a string value. (LRM 9.5.3)

The contents of ts may have been obtained by GetTimeStamp, or may alternatively have been returned by a library routine or set individually by the program. Field TimeValid must be true.

trim

Kind: Built-in function (EP standard)

Group: String handling

Format: `trim(sval)`

Returns a string-type result obtained from string value `sval` by removing any trailing spaces. A particular application is removal of space-padding from a fixed-string. (LRM 9.1.4)
See also ltrim.

trunc

Kind: Built-in function (all levels)

Group: Mathematical/Numeric

Format: `trunc(x)`

Function trunc takes a real argument x and returns an integer which is the integral part of x after truncation towards zero. See also round. (LRM 9.3.4)

unbind

Kind: Built-in procedure (EP standard)

Group: File handling

Format: `unbind(f)`

Procedure unbind breaks an association between variable `f` and an external entity (normally a file). It is not an error if the variable is not already bound. (LRM 9.2.3.5) See also bind.

unpack

Kind: Built-in procedure (all levels)

Group: Miscellaneous

Format: `unpack(pkd,unp,i)`

Move successive elements from an array of PACKED elements to an array of equivalent non-packed elements, starting at index i. (See LRM 9.7.1)

update

Kind: Built-in procedure (EP standard)

Group: File handling

Format: `update(ntf)`

Parameter `ntf` must be an indexed file which is in generation or update mode. The current contents of the buffer variable `ntf^` are written to the external file without changing the position. The recommended method of updating an element of an existing file is to call SeekUpdate to position the file, modify the contents of the buffer variable, and call update to replace the element in the external file. See SeekUpdate, SeekWrite. (LRM 9.2.4.3)

UpperCase

Kind: Built-in function (local)

Group: Character operations

Format:
```
UpperCase(ch)
UpperCase(str)
```

Parameter `ch` may be a character or wide character, parameter `str` is a string expression; the result is same type as the parameter. A lower-case letter is changed to the upper-case equivalent, any other character being returned unchanged. A string is returned with any lower-case letters changed to upper-case equivalents.   See also LowerCase. (LRM 9.7.8)

wchr

Kind: Built-in function (local)

Group: Character operations

Format: `wchr(i)`
`wchr(ch)`

Function wchr returns the Unicode character corresponding to the integer value i (i >= 0), or the 8-bit character ch. Result is wchar type. See also chr. (LRM 9.3.7)

WideToStr

Kind: Built-in function (local)

Group: String handling

Format: `WideToStr(wstr,svar)`

This function takes a Unicode string wstr, converts to 8-bit characters, and stores the result in string svar. It employs an API call, and returns a Boolean result which is true provided no error was reported from the call. If no equivalent character is available for an individual Unicode character, a default is substituted; this is not an error condition. (LRM 9.1.11)
See string, widestring.

write

Kind: Built-in procedure (all levels)

Group: File handling

Format: `write([txf,] e1[:w[:p]] [,e2 ...])`
`write(ntf, e1 [,e2 ...])`

(a) Write one or more expressions e1, e2 ... to textfile txf. If no file is specified, standard output is substituted. The file must be in generation (output) mode. Conversion of values to external representation is performed automatically, see Textfile output, writeln.

(b) Write one or more expressions e1, e2 ... to non-text file ntf. Expressions must be assignment compatible with file element type. The file must be in generation (output) mode.
(LRM 9.2.5.1)

writeln

Kind: Built-in procedure (all levels)

Group: File handling

Format: `writeln[(txf)]`
`writeln ([txf,] e1[:w[:p]] [,e2 ...])`

Procedure writeln inserts an end-of-line mark in the output textfile txf; if no file is specified, standard output is substituted. There may be parameters, which obey the same rules as in procedure write, and are processed before the end-of-line mark is inserted; that is, writeln(e1,e2) is equivalent to write(e1,e2) followed by writeln. See Textfile output.
(LRM 9.2.5.2)

writestr

Kind: Built-in procedure (EP standard)

Group: String handling

Format: `writestr(str, e1[:w[:p]] [,e2 ... ])`

The values e1, e2 ... are converted to external format as for textfile write, and the result is placed in the string str. Conversion processes are described in Textfile output. (LRM 9.1.7)

Textfile input

In textfile read, readln and readstr operations, the parameters v1, v2 etc can be variable accesses of type char, Boolean, enumerated, fixed or variable string type, integer, real or shortreal.

Type char obtains one character from the file or readstr source string.

Reading into a Boolean or enumerated type variable skips any leading spaces and is terminated when an input character is encountered that is not alphanumeric or underscore (typically, a space, comma or end-of-line). The input must match one of the identifiers in the enumeration, for Boolean these are 'false' and 'true'; the case of letters is not significant. An unrecognised identifier produces the undefined value for the type, which raises an exception if assignment range checks are requested.

Reading into a string takes characters from the source until the destination is full, or end-of-line or end of source string is reached.

Reading into a numeric variable skips any leading spaces and is terminated when the input ceases to conform to the format of an integer or real number as the case may be (typically, when a space, comma or end-of-line is encountered). Note that real numbers can be in certain forms that are not valid as constants in source programs, for instance .25 is accepted whereas the source constant must have a leading digit. An error in the format of the input causes an exception.

Textfile output

This is a summary of the provisions for writing to textfiles, for a full description see LRM 9.2.5.3.

In textfile write, writeln and writestr operations, parameters e1, e2 etc can be values of type char, Boolean, enumerated, fixed or variable string type, integer or real. Associated with each type there is a default width:

char	1
integer	6
real	14
Boolean	6
string	c

Object-Oriented programming

There is an introductory survey of the Object-Oriented features in the Introduction to Extended Pascal, and the description here expands on that background. The software package includes a number of example programs that may be helpful.

Class types

Class types describe the types of objects and the types of references. These are referred to respectively as *object type* and *reference type*.

When an object is created, it has all the features of its class; all fields, for example, are present. The process returns a *reference value* that uniquely identifies the new object. A variable of a class type holds a reference value through which you can access the object. Reference values can be assigned or passed as parameters, much as Pascal pointer values are assigned and passed. There is an important difference, however. A pointer value must be of exactly the same type as the destination to which it is assigned or passed; the corresponding requirement for a reference value is that the object identified by the value must be a *member* of the class of the reference. Because an object is a member of all ancestor classes as well as of its own class, a class-type variable can hold a reference identifying an object of its own class, or one of any subclass. When the value identifies a subclass object, the type of the variable dictates that only the features which are present in its own class can be accessed. The object itself contains the additional features of the subclass, but these are not “visible” through the ancestor-type reference.

For example, if a class `Vehicle` is defined, together with subclasses `Car`, `Truck` and `Motorcycle`, a variable of type `Vehicle` can contain a reference to a `Vehicle` object, or to a `Car` object, a `Truck` object, or a `Motorcycle` object. Based on this variable, any features of the `Vehicle` class (such as a `NumberOfWheels` field, say) can be accessed, but even when the reference identifies say a `Car` object, the features particular to the `Car` subtype cannot.

The visibility of features in the object may be further restricted if the type of the variable is a view of a class rather than a complete class. A view is introduced specifically to hide certain features of the class on which it is based, and allow access to others. A variable whose type was defined as a view can hold a reference to an object of the base class, or of any subclass of the base class. It only provides access, however, to the subset of features which the view defined to be visible.

The relationship between objects and references is expressed in terms of class type and reference type as follows.

> A class type consists of an object type and a reference type. Every reference type is based on an object type; there may be a number of reference types based on the same object type.
>
> A `CLASS` definition introduces a new object type and a new reference type. A `VIEW` definition introduces a new reference type based on an existing object type.
>
> The kind of class (concrete, property or abstract) applies to the object type, and cannot be changed by a view definition.
>
> An object is created by naming a class type. The class may have been introduced in a class definition or a view definition; in either case the object is created with the object type. The object is then identified by a reference value which is allocated at the time of creation.
>
> A variable of a class type possesses the reference type, and holds a reference value identifying an object (or the value `Null`). The reference type may or may not allow access to all components of the object.

In general usage, class-type variables are declared having a concrete class or a view of a concrete class. Nevertheless, it is possible to have a variable (or field or parameter) of a property class type. No object can be created with such a type, but when the property is inherited by a concrete class, an object can be created, and the identifying reference value can be assigned to the property-type variable. The variable allows access to just those features of the object that were inherited from the property class.

Class definitions

A class definition sets out a common structure for a group of objects, and the services available to those objects. It specifies the name and kind of class, its inheritance, fields that each object will contain, and headings of methods, constructors and destructors.

A class definition can appear in a type definition part of a program, module heading or module block. Class definitions cannot be introduced within procedure or function declarations. A class definition must introduce a new named type.

A property or abstract class has the word PROPERTY or ABSTRACT in its definition; other classes are concrete classes. A property class defines a set of related components which can be inherited by other classes, making the basis for multiple inheritance. An abstract class is a placeholder in the concrete hierarchy, and defines common contents for descendent classes. Property and abstract classes have an object type, but this defines the contribution that they make to an object rather than a complete object. Objects can only be created from concrete classes.

The following example shows a sequence of class definitions.

```
TYPE  Conc1 = CLASS            { Root implied }
                cf1,cf2: integer;
              END;
      Prop1 = PROPERTY CLASS   { no ancestors }
                pf1,pf2: char;
              END;
      Prop2 = PROPERTY CLASS   { no ancestors }
                pf3: Conc1;
              END;
      Prop3 = PROPERTY CLASS (Prop2)
                FUNCTION ready: Boolean;
              END;
      Conc2 = CLASS (Conc1,Prop1,Prop3)
                cf3: ARRAY [1..10] OF real;
              END;
```

An object of type `Conc2` contains the inherited fields `cf1`, `cf2`, `pf1`, `pf2` and `pf3`, together with the field `cf3` introduced in its own component list. Type `Conc2` has also inherited the components of `Root` (via `Conc1`) and the function `ready` from `Prop3`. All the definitions are classes (rather than views), and the reference types make visible all the components of each class.

A class definition can be "deferred", to provide the possibility of introducing references of the new class type as fields, parameters, or function results. To do this, the name is first introduced thus:

```
      TYPE  newclass = CLASS .. END;
```

If the class is to be a property or abstract class, the word `PROPERTY` or `ABSTRACT` must precede `CLASS`. Later in the same type-definition-part the actual definition must be provided. The class name cannot be used before its definition is complete unless a deferred definition precedes the use.

A class definition may specify an inheritance list, that is, a list of class names from which the new class directly inherits fields, methods, constructors and destructors. In practice, most class definitions include an inheritance list; if present, it is contained between parentheses. A class name in the inheritance list may have been introduced in a class definition or a view definition. The new object type contains all the components of the object types of its ancestors; the new reference type makes visible the components of the new object type which were visible in the reference types of its ancestors. Inheritance provides the means of defining an extension or refinement of an existing class.

A concrete or abstract class can inherit from not more than one concrete or abstract parent. If a new concrete or abstract class has no inheritance list, or names only property classes, it inherits automatically from the `Root` class. A property class can inherit only from other property classes. Any class can inherit from zero or more property classes, provided that these classes have no common ancestor. It cannot inherit from itself, more than once from the same class, or from two classes with a common ancestor. The inheritance model implied by these rules is one in which multiple inheritance is restricted to separately-rooted trees; the example above illustrates a valid inheritance pattern.

After the inheritance list, the class definition lists any new features which are to be added to the type. There may be fields, which will form part of each object, or methods (procedures and functions), constructors and destructors. Methods, constructors and destructors are collectively called *procedural items*. Fields are declared like variables or record fields; a class procedural item is represented by its heading, with parameter list and (for a function) result type. The list can also contain headings of procedural items which are inherited from an ancestor class and overridden in the new class.

Class membership

An object is a *member* of its object type, and of all ancestors of its object type. Membership determines the validity of certain operations, and can be tested using the `IS` operator.

Names of class features

The names of new features (fields and procedural items) in a class component list must not introduce possible ambiguities. Specifically, a name must not have the same spelling as (a) the name of the new class, (b) the name of any ancestor class or view, (c) the name of any visible inherited component, or (d) the parameter `self`.

The names in a class follow the same scope rules as field names in a record type; briefly, they are local to the class, and the names visible in the reference type come into scope when a variable of the type is accessed. The scope can also be opened by a WITH statement quoting a reference to an object.

Overriding inherited definitions

When a method, constructor or destructor is inherited from a parent class, its actions can be overridden. Any procedural items can be overridden; the effect of overriding is to substitute a new declaration which redefines the action taken when the item is activated. The new version forms part of the object type of the new class, and is inherited by descendant classes.

The name of the item is included in the component-list of the new class, followed by the directive `override`. The signature of the item (parameter list and function result type) cannot be changed when overriding; it can be omitted or repeated, and if repeated must agree with the original.

```
        FUNCTION ready: Boolean; override;
```

The override directive is essentially a confirmation that the name was intended to match an existing name, and that the repetition is not an error. The new declaration (body) must appear in the program block, or in the block of the module in which the new class is defined.

Abstract methods

A method, constructor or destructor introduced into a class type may be marked `ABSTRACT`, for example

```
     FUNCTION Precedes (x: Root): Boolean; ABSTRACT;
```

An abstract procedural item is permitted only in a property class or an abstract class. An inherited non-abstract method cannot be made abstract in a descendant class. No implementation is provided for an abstract method in the class in which it is defined, but must be provided in a descendant class at least by the point when a concrete class is defined. A concrete class cannot contain an abstract method which has not been overridden and provided with an implementation.

View definitions

A view definition introduces a new reference type based on an existing object type, which must already be fully defined (that is, not deferred). A view definition can appear only in a type definition part of a program, module heading or module block. It can specify visibility by means of class names, component names, or both. If there are class names, the list is contained between parentheses. A class name must be either an ancestor class of the base class or a previously defined view of the same base class. (The latter allows nested views of the base class to be built up starting with the most restricted.) All visible components of the base class that were visible in the named class are included in the new view. A list of components names individual components that are to be included in the view, which must be visible components of the base class. The new view is formed from the OR of the two lists.

A view of a concrete class, to be useful, is likely to include a constructor. The following example shows a view based on the class Conc2.

```
TYPE  View2 = VIEW OF Conc2 (Prop1)
                Create,cf3
               END;
```

The list of classes simply names the ancestor type `Prop1`, the list of components adds the constructor `Create` and the field `cf3`. The view contains the fields `pf1` and `pf2`, inherited from `Prop1`, constructor `Create` inherited from `Root`, and field `cf3` from `Conc2` itself. The other fields and the function `ready` are not visible, nor are the other items inherited from `Root`. Field names `pf1` and `pf2` could have appeared in the component visibility list, but would make no difference to the contents of the view.

Like a class definition, a view definition can be "deferred", and the notation for introducing it is similar:

```
TYPE  newview = VIEW OF baseclass .. END;
```

Later in the same type-definition-part the actual definition must be provided.

A class definition may name a class view as an ancestor. The object type of the new class will include all the components of the object type upon which the view was based, and the reference type of the new class will contain the components that were visible in the view; the reference type may not (indeed, probably will not) make visible all the components of the new object type. When a class with restricted visibility is quoted as an ancestor or as the base of a new view, components cannot be made visible that were not already in its reference type; that is to say, a new subclass or view cannot recover visibility previously removed.

A field name in a view definition may be made PROTECTED, and any access to the field through a reference of the view-type must then not "threaten" it. If the view is exported, either by name or as the type of a reference, an importer can obtain the current value but not modify the field. The protection is inherited if the view type is named as an ancestor. Because this precludes an overriding method from making any change, protection should be applied with care.

Fields of objects

Fields introduced in a class definition form part of the object type of the class. They are *instance variables*, that is, they are reproduced each time that an object of the type is created. The type of a field is given by a type denoter, which allows any named type, new type, or discriminated schema, but not a new schema definition or new class definition. A deferred class definition is permitted as the type of a field. A field defined with a class type holds a reference to an object.

The type of a field may be a type produced from a schema, and may be one whose size is not known prior to execution of the program. The layout of fields within an object follows the order in which they are introduced in the component list, and as with records there is some advantage in fields of schematic types whose size is not known until run-time being placed last.

A field in the object type of a class is inherited by the object type of every descendant class. The name of a new field also forms part of the reference type of the class in which it was introduced, and of descendant classes, but may be excluded from a view based on the original class or a descendant.

A field may be defined with an initial state (initial value), either through the initial state being associated with the type of the field or through it being given in the field declaration. The initial state is supplied automatically when an object is created, and for constant values this is the recommended way of ensuring that a new object is in a well-defined state.

Statements can access fields of an object with two notations, again similar to fields of records. The first is to name an object reference followed by a period and the name of the field. Alternatively, a WITH statement quoting an object reference brings into scope the names of all components of the object that are visible in the reference type, and they can be accessed immediately. A special case is the declaration (body) of a class procedural item, where all components of the class are immediately accessible, as though an implied WITH surrounded the declaration.

Methods

A method is a procedure or function that provides a service for objects. Methods are introduced into a class definition by including their headings in the component list, and form part of the object type of the class and all descendant classes. For a non-abstract method there must be a declaration in the program block or module block, as described below.

Method heading

Method declaration

Method activation

Method heading

The method heading includes the parameter list (if any) of the method, and the result type of a function. Value and variable (`VAR`) parameters may be of any named type that is in scope, including class types, or may be of conformant array type. A method may have a procedural or functional parameter, but only conventional procedures and functions can be passed as actual parameters; a method cannot be passed as an actual parameter, either to another method or to a conventional procedure or function. The result type of a method function may be any named type that is in scope, including a class type (which returns a reference). An extra formal parameter called `self` is automatically supplied for every method, even if it has no parameter list in its heading. It is a value parameter whose type is the class in which the method is defined.

The overriding of methods was introduced earlier. An overriding version of a method is included in the object type of a new class in place of the original, and is inherited by descendant classes unless it is overridden in its turn.

Method declaration

The declaration of a method defines the actions to be performed when the method is activated. It takes the form of a procedure or function declaration in which the identifier that names the item is preceded by the class name, for example

```
FUNCTION Prop3.ready: Boolean;
```

The signature may be repeated, and will make the code more readable; the directive `override` may also be repeated where appropriate. The rest of the declaration is a procedure-block or function-block, which can introduce labels, constants, types and local variables as in conventional procedure or function declarations.

When a method is activated, as described below, an object reference is automatically passed in place of the formal parameter `self`. Within the declaration of the method, `self` provides a reference value, and for most purposes may be used just like any other value parameter of class type. It can be assigned, compared or passed as an actual parameter in calls of other methods. It may not, however, be used in a destructor activation.

Within the statement part of the method block, the names of all visible components of the class in which the method is defined are in scope, and can be accessed without qualification. Any fields are those of the object through which the method was activated. The scope which makes available the component names encloses the whole method declaration, so it is possible for a locally-declared name, such as the name of a local variable, to obscure a component name, but this situation can normally be avoided because the definition of the class type is known when the method declaration is coded; if it arises the local name takes precedence, and the component must be accessed as `self.componentname`.

The declaration of a method may include activation of visible methods inherited from ancestor classes. A particular and quite common use when overriding is to activate the inherited version at the start of the new implementation. This ensures that any changes made to the inherited version are carried through automatically. The version of a method from an immediate parent class is activated as "`INHERITED methodname`"; more generally, the form "`classname.methodname`" can be used for more distant ancestors.

A method declaration may include nested procedures or functions. The nested declarations can use the `self` parameter, and the object components are accessible, just as in the immediate method block.

Method activation

A method can be activated only through a reference, either explicit or implied. It cannot be called directly from non-object code. An activation with a specific reference takes a form similar to access to a field, that is, an object reference followed by a period and the name of the method. If the method has a formal parameter list, actual parameters must be supplied as in the call of a conventional procedure or function. If the method is a procedure the activation is a form of procedure statement; if it is a function the activation is a form of primary. The reference through which the method was activated is supplied as the actual corresponding to the formal parameter `self`.

The `INHERITED` and ancestor-name forms are valid only within the body of a method, when invoking the versions of methods inherited from ancestor classes as described in the previous section. The `self` supplied to the method making the call is passed on to the called method as its `self` parameter.

Constructors and destructors

A constructor defines actions to be performed when an object is created; a destructor defines actions to be performed when an object is removed. Although new destructors can be introduced, their use is not recommended; where possible, any special actions needed should be undertaken by overriding the destructor `Destroy` inherited from Root. A constructor or destructor is introduced into a class definition by including its heading in the component list; it forms part of the object type of the class and all descendant classes. A non-abstract constructor or destructor must have a declaration in the program block or module block, as described below.

Constructor and destructor headings

Constructor and destructor declarations

Constructor activation

Destructor activation

Construction of objects

Destruction of objects

Constructor and destructor headings

The heading for a constructor or destructor begins `CONSTRUCTOR` or `DESTRUCTOR`, and includes the list of formal parameters, if any. Value and variable (VAR) parameters may be of any named type that is in scope, including class types, or may be of conformant array type. A constructor or destructor cannot be passed as a parameter. An extra formal parameter called `self` is automatically supplied for every constructor and destructor, even if it has no parameter list in its heading. This is a value parameter whose type is the class in which the item is defined. A heading may be marked `ABSTRACT` or `override`; an overriding version of a constructor or destructor is included in the object type of a new class in place of the original, and is inherited by descendant classes unless it is overridden in its turn.

Constructor and destructor declarations

The declaration of a constructor or destructor defines actions to be performed when objects are created or destroyed, which will include activating any inherited constructors or destructors.

The identifier that names the constructor or destructor is preceded by the class name and a period, as in declaration of methods, and the signature can optionally be repeated. The rest of the declaration is a procedure-block, which can introduce labels, constants, types and local variables as in conventional procedure declarations.

Within the statement part of the procedure block, the names of all visible components of the class in which the constructor or destructor is defined are in scope, and can be accessed without qualification. The only exception to this rule is when a component name is “masked” by a parameter or local variable name having the same spelling, as described in 13.4.2.

Again as in method declarations, the parameter `self` provides a reference value, and may be used as a value parameter of class type. Its type is that of the class in which the constructor or destructor was defined.

The activation of inherited constructors and destructors is described below. The version from an immediate parent class can be activated as INHERITED itemname, or for a version from a specific ancestor the form “classname.itemname” can be used.

A constructor or destructor declaration may include nested procedures or functions. The nested declarations can use the `self` parameter, and the object components are accessible, just as in the immediate procedure block.

Constructor activation

A constructor can be activated either by a *constructor-access* (which creates a new object) or by a *constructor-statement*. A constructor-access is a form of primary; it is accompanied by obtaining the space for a new object and setting the initial states of fields, and returns a reference value which uniquely identifies the newly-created object. A constructor-statement is used in the body of a constructor to activate a constructor in an ancestor class.

A constructor-access is introduced by the name of a concrete class, and the object is created having the object type of that class.

```
VAR   Cref: Conc2;
  ..
  Cref := Conc2.Create;
```

The constructor identifier must be visible in the reference type of the class. A constructor defined in a property class can be activated only by means of a constructor-statement in a descendant. The INHERITED and ancestor-name forms allow reference respectively to the immediate parents and to ancestor classes more generally, as in the activation of inherited methods (see Method activation).

When a constructor is activated, any actual parameters are evaluated, and the object then temporarily adopts the type in which that version of the constructor was declared. Methods activated by the constructor will be the versions appropriate to the adopted type. This is an exception to the general rule concerning behaviour of objects, and is designed to ensure that the new object can be initialised safely. An overriding definition of a method might refer to parts of the object that were not yet ready, with unfortunate consequences

Destructor activation

A destructor is activated through a reference to an object which is to be removed from its class. For instance, if `Cref` contains a reference to an object, the following statement will cause the object to be destroyed.

```
        Cref.Destroy;
```

The object-reference in an initial destructor activation must be of a concrete class type. A destructor defined in a property class can be activated only by means of a destructor-statement in a descendant class. The INHERITED and ancestor-name forms are for reference respectively to the immediate parents and to ancestor classes more generally.

The initial destructor may activate destructors in ancestor classes, and these are called *continuing* destructor activations. During a continuing destructor activation, the object temporarily adopts the type of the destructor, and this change governs the activation of methods, as described above for constructors.

Construction of objects

It is recommended that initial state specifiers should be employed where possible to initialise fields of a new object, and that constructors be written only to perform other kinds of preparation, such as initializations that involve run-time values, linking the new object to a chain, or incrementing a global count.

If a class inherits non-abstract constructors from more than one parent, a non-abstract constructor must be included in the class. This will often be an overridden version of one of the inherited constructors, but may be new. It should first activate the inherited constructors, so that the object is properly defined, and then perform any specific tasks. In the common situation of a new concrete class inheriting a property class, a version of `Create` will always be inherited from its concrete parent, and if the property class also contains a constructor a new constructor must be provided. Typically, this will be by overriding the inherited `Create`.

Destruction of objects

As a rule, it is simpler to destroy an object than to create one, but there may be need to detach the object or to decrement a count, and such operations are appropriately performed by a destructor. Destruction should normally proceed in the reverse of the order of construction, with calls of inherited destructors made after any specific actions have been performed.

Once an initial destructor activation has commenced, the state of the object is not reliable, and any use of the reference value that identifies the object (for example by another thread) is an error. On completion of the initial activation, the object is removed from the class.

Predefined entities

There are four predefined entities associated with objects, a constant (Null), two class types (Root and TextWritable) and a function (Copy).

Null

The identifier Null represents a constant that is compatible and assignment-compatible with any variable of class type, and represents no object. Every reference variable or field is automatically given the value Null as its initial state. A reference may be compared against Null, and Null may be assigned or passed as an actual value parameter. An exception is raised if an attempt is made to use a reference which holds the value Null to access a component of an object.

There is a clear analogy with NIL and pointer types, and the analogy is helpful, but NIL and Null are not interchangeable.

Root

The identifier Root denotes an abstract class that is an ancestor of every user-defined concrete or abstract class. If the definition of a concrete or abstract class does not specify a concrete or abstract parent in its inheritance list, direct inheritance from Root is implied. If the definition of the Root class was written in a program it would appear as follows.

```
TYPE  Root = ABSTRACT CLASS .. END;
      Root = ABSTRACT CLASS
               CONSTRUCTOR Create;
               DESTRUCTOR Destroy;
               FUNCTION Clone: Root;
               FUNCTION Equal(R: Root): Boolean;
             END;
```

The preliminary deferred definition is needed because of the two applied occurrences within the actual definition. There is a slightly artificial aspect of this presentation, because it is the one abstract class with no inheritance list that does not inherit from Root. The components Create, Destroy, Clone and Equal are described below.

Create A constructor-access quoting `Create` as the constructor name obtains space for the new object, sets any defined initial states, invokes the version of `Create` appropriate to the type, and returns a reference to the object. The version of `Create` in the predefined `Root` class does nothing further, but is available to be overridden when the process of creating a new object involves actions (such as counting the number of live objects, say).

When overriding a constructor, the first statement in the declaration should normally be a constructor-statement to activate the inherited version. This ensures that at each stage the new object is in a correct state for any further actions to be performed.

Destroy An initial destructor activation quoting `Destroy` as the destructor name invokes the version of `Destroy` appropriate to the object type, and on completion removes the object. The version of `Destroy` in the predefined `Root` class does nothing except remove the object, but is available to be overridden in descendant classes when there are actions needed. The declaration of an overriding version of a destructor should normally conclude by activating the inherited version.

Clone There is a predefined function `Copy` for reproducing an object. The functional method `Clone` in the `Root` class returns the result of applying `Copy` to the parameter `self`. It is available to be overridden in descendant classes when there are actions associated with correctly establishing the new object, much as when `Create` is overridden. The recommended approach is that the first statement in an overriding version of `Clone` should activate `INHERITED Clone`. After calling the inherited version, the new object is in a suitable state for actions appropriate to the inheriting class to be performed.

Function `Clone` as inherited from `Root` cannot be activated if the object type includes a file, or a structured type containing a file. (See assignment compatible.)

The following is an outline declaration of an overriding version of `Clone` in a concrete class `newclass`.

```
FUNCTION  newclass.Clone: Root; override;
  VAR  x: newclass;
  BEGIN
    x := newclass(INHERITED Clone);
    ...
    Clone := x;
  END {newclass.Clone};
```

It is assumed here that the reference variable x of type newclass is needed by the statements shown as an ellipsis, whose actions were the reason for introducing the overriding version in the first place. If they do not require a reference to the new object, x can be omitted and the outline can be simplified:

```
FUNCTION  newclass.Clone: Root; override;
  BEGIN
    Clone := INHERITED Clone;
    ...
  END {newclass.Clone};
```

Equal The functional method Equal as defined in Root compares the references R and self, and returns true if they identify the same object. In this form it is therefore equivalent to a simple comparison of the two references, but can be overridden in descendant classes to modify this behaviour, for instance by comparing fields.

TextWritable

The identifier TextWritable denotes a property class containing two procedural methods. If the definition of TextWritable were to appear in a program, it would look like this.

```
TYPE  TextWritable =
          PROPERTY CLASS
            PROCEDURE ReadObj (VAR f: text);
            PROCEDURE WriteObj (VAR f: text);
          END;
```

As predefined, the procedures ReadObj and WriteObj do nothing. The intended usage is for TextWritable to be inherited and one or both of the procedures to be overridden. The overriding version of ReadObj should read an object from its parameter f and that of WriteObj should write an object to its parameter f. When an inherited class includes TextWritable among its ancestors, the new version calls INHERITED ReadObj (or WriteObj), and adds statements to read or write any fields added in the new class.

Copy

`Copy` is a predefined function with one parameter, a reference to an object. The object type of this reference must be a concrete class type. `Copy` creates a new object of the same object type, and copies the fields of the original to the new object. It then returns a reference to the new object, the type of this reference being that of the original parameter.

```
NewRef := copy(ObjRef);
```

The parameter to `Copy` must be one whose fields are all of assignable types, in particular they must not include any file or structure containing a file. It is an error if any field is undefined at the time `Copy` is invoked.

Note that there is another predefined function `Copy` that takes a string as its first parameter. Provided they are used correctly there is no risk of confusion.

Working with objects

Much of the preceding description has been concerned with defining classes, and implementing constructors and methods. To use a class, however, you should only be aware of the functionality that the originator wished to make available, and most often this will be in terms of creating objects, and invoking methods to perform actions on or with them. It is the purpose of this part of the description to bring together the relevant details, and show how they link into conventional Pascal. When the original definition does not fully meet your requirements, of course, you may consider defining a subclass to add to the capability.

Primaries An access to a field of an object, and the activation of a constructor or a functional method, can be used as forms of primary within expressions.

Compatibility The following rules on compatible and assignment-compatible types relate to classes.

> All class references, and `Null`, are compatible in comparisons.
>
> A value of a reference type T2 is assignment-compatible with a reference type T1 if the base type of T2 is the same as the base type of T1, or inherits from the base type of T1.
>
> The value `Null` is assignment-compatible with all reference types.

The rule of assignment-compatibility of reference types permits assignment to a property-class reference provided that the object inherits that property.

Assignment An assignment statement attributes a reference value identifying an object, or the value `Null`, to a variable possessing a class type. The assignment-compatibility requirements in the previous paragraph must be observed. Assignment of a reference value does not affect the object itself.

An access to a field of an object may be the destination of an assignment, provided the type of the field is assignment-compatible with the value being assigned. The fact that it is a field of an object does not affect the operation.

Reference coercions A class-type variable can hold a reference value which identifies an object of its own type or of any subtype. The assignment-compatibility rule allows "up-level" assignment to an ancestor type. However, situations arise in which a variable is known to hold a subtype reference, and a *reference coercion* allows the type of the variable (the ancestor type) to be transformed to this subtype. A run-time check is performed, and an exception is raised if the coercion is not valid.

Returning to the `Vehicle` hierarchy introduced earlier, if a variable `v` of type `Vehicle` which is known to hold a reference to a `Truck` object, the reference can be assigned to a variable of type `Truck`, or passed as a parameter of type `Truck`, by writing `Truck(v)` . That is, you *coerce* the reference by writing the type name followed by the reference in parentheses. If the reference was not in fact to a `Truck` object, but to a `Vehicle` object, or say a `Car` object, the coercion would raise an exception.

Comparison The relational operators "=" and "<>" can be used to compare reference values. The result of "=" is `true` if both operands contain the value `Null` or both identify the same object, otherwise the result is `false`. The result of "<>" is the logical negation of "=". Because a reference value uniquely identifies an object, equal values identify the same object. See also the predefined function `Equal`.

Parameters A reference value can be passed as an actual value parameter corresponding to a value formal parameter of class type, provided that the assignment-compatibility requirements are met. As with other types, the actual parameter corresponding to a variable (VAR) formal parameter must be a variable-access of the same type as the formal parameter.

A field of an object can be passed as an actual value or variable parameter provided the type of the field is appropriate to the kind and type of the corresponding formal parameter. The fact that it is a field of an object does not restrict its use as an actual parameter; however, when passed as an actual VAR parameter it is an error if the object is destroyed before return from the call.

IS operator The class membership operator IS allows you to check the type of an object identified by a reference. The left operand is the reference and the right operand is a class name. The operator returns true if the object is a member of the class (that is, the type of the object is the same as, or inherits from, the class). Referring to the definition of class Conc2, an object of type Conc2 is a member of Root, Conc1, Prop1, Prop2, Prop3 and Conc2, but an object of class Conc1 is a member of Root and Conc1 only.

It will be seen that this test is closely related to the run-time check involved in a reference coercion, and can be used to check in advance the validity of a coercion, particularly if run-time exceptions are to be avoided. Alternatively, you may prefer to guard the coercion with a TRY statement.

WITH statement A WITH statement that contains a reference to an object brings into scope the identifiers of all fields, methods and destructors that are visible in the type of the reference. Fields are accessed, and methods and destructors activated, by quoting the component name only. In the activation of methods and destructors, the value of the reference in the with-element is passed as self.

As with records addressed by pointer dereference, the object-reference is evaluated at the point of the WITH statement and is not affected by operations performed by the dependent statement. It is an error if the object is destroyed during execution of the dependent statement.

Export and import of classes

Like other types, class types and views can be exported by naming them in an export-list. Exporting a view allows the originator to restrict access by importers to those components of a class that are visible in the view. Some special considerations applying to export, import and use of classes are described below.

Ancestor classes

Export of a class automatically results in the definitions of all ancestor classes being included in the interface, and this may represent a significant amount of information. Another important consideration is that while the contents of these ancestor classes are automatically included, their names are not, but must be specified in the list if they are to be available to an importer. Without these names, a user can import the class, define a subclass, and refer to INHERITED methods, but not to methods in named ancestor classes. The user also cannot define subclasses based on ancestors. While export of a view gives the exporter control of the range of components that are visible, the inclusion or exclusion of ancestor names should also be kept in mind as significantly affecting the uses to which an imported class can be put.

Feature renaming

When a class or view is exported, it can be renamed in the same way as other constituents. A notation is also available for renaming individual features of the class or view, by means of *feature renaming*. Feature renaming can only be used when the constituent is a class type.

An identifier introduced by feature renaming is subject to the rules that apply to the names of new components in a class definition, and in particular must not be the name of an ancestor class or of a visible feature of the class. The new name identifies the component in the exported class and in any subclass derived from it.

Feature renaming on export provides a means of avoiding some possible sources of difficulty when classes which were originated independently are to be inherited. For example, a module can be introduced which simply imports a class and adapts it by re-exporting with one or more components renamed. However, there are a two points to beware of. The first is that when a component has been inherited by two or more classes, renaming it in one does not affect its name in any others. The other is that the renaming does not modify the original class. When a class is exported with feature renaming, imported, and a subclass formed from it, the implementation of any overridden method must refer to the INHERITED version quoting the name by which it was originally known. This implies that the importer has knowledge of the names in the exporting module, so renaming of methods should be avoided where possible.

Violations, errors and exceptions

To err is human, they say, and even programmers sometimes make mistakes. This section gives an outline description of the ways in which the language can help you avoid mistakes, and the implementation can help you to eliminate those that still get through.

Pascal contains features which are intended to assist in the production of robust programs. (It is not unique in this, but it has generally been one of the leaders as programming languages have developed.) Some references to this aspect have been made earlier, and to summarise here are a few guidelines:

> Make the names you devise work for you. Meaningful names are a big help, both to others, and to yourself when you revisit your work later.
>
> Introduce named constants, particularly when they are used more than once.
>
> Define enumerated types for collections of related items (days of the week, colours of the rainbow, fonts, kinds of locomotive, and so on). These groups of names are kept distinct from one another and from other items such as variables.
>
> Use procedures and functions to break up code into pieces of manageable size, as well as to avoid duplication. Procedures can "encapsulate" subsidiary actions, and allow attention to be concentrated on different levels of detail.
>
> If the situation justifies it, employ modules as another intermediate level, or when parts of a program may be re-used. You can export named constants and types from a module, as well as variables, procedures and functions.

This implementation of Extended Pascal includes a range of facilities for detecting mistakes, both at compile time and when a program is run. Some checks are always performed, others are specially requested by means of compiler options. The standard uses the word violation for an illegal construct that an implementation must detect, and the word error for one that it may not be feasible to find (typically, one that involves run-time checking). The majority of violations are detected during compilation, a few involve run-time checks.

Typical examples of "errors" are array index values outside the bounds with which the array was declared, or references to character positions in a string beyond the current contents, which will generally produce nonsense values. If you suspect such problems you can ask for checks to be incorporated in your program by the compiler – see the Compiler Options in the Workbench Build menu for instance. Some kinds of error are always checked, an example is attempting to read past the end of an input file. Other, more subtle, kinds of error arise from using a copy of an out-of-date pointer, mistakes in the use of variant records, or reference to a variable to which no value has yet been given, for which checking is optional. Overall, this implementation is unusually well provided with optional checks.

Besides these general checks, there is a means of defining specific checks at individual points in your code. The `assert` procedure is given a condition which you believe to be true, and you can request the compiler to generate a check at that point, otherwise the assertion is ignored. As a simple example, you could assert when entering one of your own procedures that a pointer parameter is not `NIL`, or a string contains at least four characters. These are assumptions you made when coding your procedure, and if they are not satisfied the call is in some sense incorrect.

The failure of any run-time check raises an exception. Generally, an exception is notified to the user of the program as a message, which includes details of the kind and location of the failure. During testing, you can use the information in the message to track down the mistake that gave rise to it. However, in a production program you can choose to intercept particular exceptions, such as those arising from input of faulty data. You do this by means of `TRY` statements.

Occasionally it may be convenient to raise an exception of your own, which you can do with the `RaiseUser` procedure. This is not as a rule appropriate in simple programs, but if in a more complex application a situation arises in a nest of procedure calls that cannot be handled at that point, an exception can be raised that is intercepted further back, where some general escape can be provided.

Following is the classification of causes of exceptions, which appears in notification messages and is used in the coding of `TRY` statements. There is a general division between “language” exceptions resulting from run-time checks on Pascal usage, “system” exceptions such as implementation, and “user” exceptions originating from calls of the `RaiseUser` procedure. The language exceptions are grouped into various categories, and within these categories are more specific causes.

Here is the outline classification; the full list is in the following section.

- Language
  - Access
  - Assignment
  - BindUnbind
  - Control
  - File
  - Format
  - Nonstatic
  - Numeric
  - Variants
  - Other
- System
- User

Exception classes

```
ExcepLanguage

    ExcepAccess
        ExcepArrayIndex
        ExcepStringIndex
        ExcepSubstring
        ExcepPointer
        ExcepField
        ExcepAccessViolation
        ExcepRefUndefined
        ExcepRefExists
    ExcepAssignment
        ExcepOrdinalValue
        ExcepStringOverflow
        ExcepSetOverflow
        ExcepStructuredValue
        ExcepClassError
    ExcepBindUnbind
        ExcepNonbindable
        ExcepAlreadyBound
        ExcepFileType
        ExcepIncomplete
    ExcepControl
        ExcepCASE
        ExcepGOTO
    ExcepFile
        ExcepUndefinedFile
        ExcepIncorrectMode
        ExcepUndefinedBuffer
        ExcepDirectAccess
        ExcepReadOnly
        ExcepAtEndOfFile
    ExcepFormat
        ExcepInputInteger
        ExcepInputNumber
        ExcepTotalWidth
        ExcepFracDigits
        ExcepReadstr
        ExcepWritestr
    ExcepNonstatic
        ExcepTypeMismatch
        ExcepSubrange
        ExcepEmptyDomain
        ExcepActualDiscriminant
        ExcepParameterType
        ExcepResultType
        ExcepNotConformable
    ExcepNumeric
        ExcepIntegerOverflow
        ExcepFloatingOverflow
        ExcepFloatingUnderflow
        ExcepIllegalValue
        ExcepPrecision
```

- **ExcepVariants**
  - ExcepFixedVariant
  - ExcepSelectorValue
  - ExcepDisposeVariant
- **ExcepOther**
  - ExcepDate
  - ExcepResultUndefined
  - ExcepLocalExtension

ExcepSystem

- ExcepStackOverflow
- ExcepAllocationFailure
- ExcepOSFailure
- ExcepInitialization
- ExcepTermination
- ExcepThreads
- ExcepLocalLimit

ExcepUser